Universidad Complutense de Madrid
FACULTAD DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA

SISTEMAS INFORMÁTICOS

ALGORITMOS PARA GRAFOS Y PROGRAMACIÓN DE PROPÓSITO GENERAL EN CUDA

Curso 2008/2009

Ezequiel Denegri Guillermo Frontera Sánchez

Dirección:

Pedro Jesús Martín de la Calle Antonio Gavilanes Franco

Resumen

Las actuales unidades de procesamiento gráfico de NVIDIA disponen de una interfaz de programación que permite utilizarlas para tareas de propósito general (GPGPU).

El objetivo de este trabajo es acelerar la ejecución de algoritmos clásicos de la teoría de grafos utilizando la interfaz de programación CUDA. Para ello, hemos estudiado la formulación y posterior codificación de estos algoritmos en el modelo de programación paralela de CUDA. En particular, los algoritmos estudiados son la búsqueda de los caminos mínimos desde un único origen, la búsqueda del árbol de recubrimiento mínimo y la búsqueda de todos los caminos mínimos entre pares de vértices.

También hemos estudiado algunos algoritmos no relacionados con la teoría de grafos, haciendo una breve exposición de aplicaciones de la GPGPU a la programación evolutiva, la visión por computador y al cómputo requerido por un motor de búsqueda.

Abstract

Current graphics processing units manufactured by NVIDIA provide a programming interface that allows them to perform general purpose computation.

The aim of this work is to speed up some classic algorithms in graph theory using CUDA programming interface. To this end, we have studied sequential formulations of these algorithms and their related code using CUDA parallel programming model. Specifically, we have studied algorithms for the single source shortest path search, the minimum spanning tree search and the all vertex pairs shortest path search.

We have also studied several other algorithms not related to graph theory as an overview of GPGPU applications to evolutionary programming, computer vision and computing performed by a search engine.

Palabras clave: CUDA, General Purpose Computing on Graphics Processing Units, GPGPU, algoritmos sobre grafos, algoritmos evolutivos

Índice

Autorización

Autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Firmado: ....
Ezequiel Denegri Guillermo Frontera Sánchez

Madrid, 3 de julio de 2009.

1. Introducción

Hoy en día nos encontramos en una situación límite en el cumplimiento de la ya conocida *Ley de Moore* sobre el rendimiento hardware. Se ha alcanzado el rendimiento pico en CPUs monoprocesadores, comenzando a utilizar los nuevos transistores disponibles para construir multiprocesadores, de doble o cuádruple núcleo.

Todo indica que se llegará a una nueva generación de ordenadores *many-core* (*muchos núcleos*) en los cuales los algoritmos secuenciales tradicionales perderán importancia frente a los algoritmos que permitan el procesamiento paralelo de datos.

Si bien pensar en este tipo de procesadores actualmente está vinculado a grandes cantidades de dinero, existe una posibilidad de bajo coste que resulta novedosa en el área de la programación de algoritmos paralelos: utilizar la enorme potencia de cálculo y el gran número de procesadores de las tarjetas gráficas (en adelante GPUs, de *Graphics Processing Units*) para llevar a cabo la ejecución de tareas no vinculadas con actividades gráficas, es decir llevar a cabo lo que se conoce como **programación de propósito general sobre GPUs (GPGPU)**.

Las tarjetas más actuales dedican la mayor parte de sus transistores a crear unidades de procesamiento más que al control de flujos de datos. En esta línea, una tarjeta gráfica de coste medio puede tener alrededor de **128 procesadores**, con capacidad de ejecutar tanto tareas inherentes al renderizado de imágenes como programas de propósito general, todo ello de forma paralela.

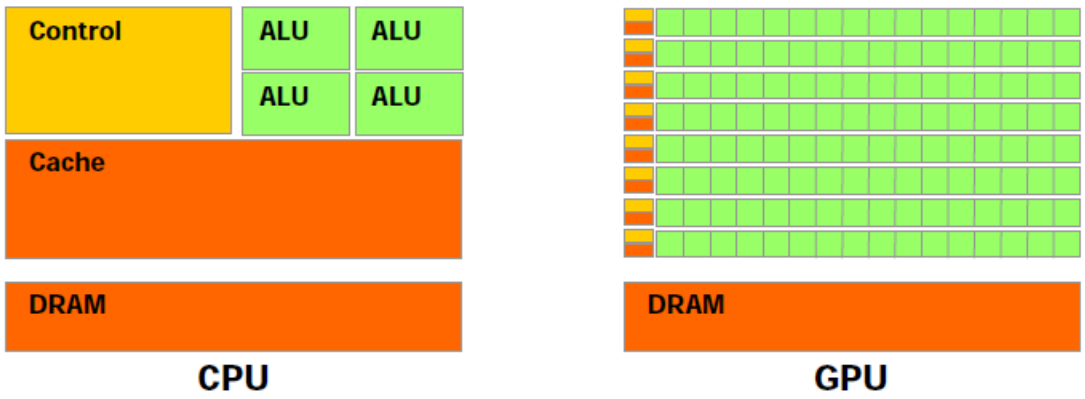


Figura 1-1. Comparativa sobre el uso de transistores.

Estas condiciones de ejecución hacen atractivo establecer una competición entre el rendimiento de una CPU (Unidad Central de Proceso, procesador clásico) y una GPU (Unidad de Proceso Gráfico). En términos generales, los algoritmos a ejecutar en una GPU van a ser versiones paralelas de los que se ejecutan en la CPU, pues es esta la forma de conseguir un mayor rendimiento de la misma.

Por otra parte, muchas aplicaciones que trabajan sobre grandes cantidades de datos se pueden adaptar a un modelo de programación paralela, con el fin de incrementar el rendimiento en velocidad de las mismas. La mayor parte de ellas tienen que ver con el procesamiento de imágenes y datos multimedia, como la codificación/descodificación de vídeos y audios, el escalado de imágenes, y el reconocimiento de patrones en imágenes. Sin

embargo existe un gran abanico de algoritmos y aplicaciones que no están vinculados con este campo, pero que son susceptibles de un tratamiento paralelo, y es en ellos en los que nos vamos a centrar; más concretamente en los que están dentro del área de los **algoritmos sobre grafos** y otros algoritmos de importancia.

Para realizar nuestro proyecto hemos usado la tecnología que ofrecen las tarjetas gráficas de NVIDIA, que cuentan con un lenguaje de programación específico, **CUDA** (de *Compute Unified Device Architecture*). Este lenguaje es C con un conjunto de extensiones que permiten la especificación del ámbito de ejecución de las funciones y el espacio de memoria en donde residen las variables declaradas. CUDA cuenta con un compilador desarrollado por NVIDIA que es capaz de generar el código binario independientemente del número de procesadores que tenga la GPU en la cual se va a ejecutar el programa.

Esta memoria se organizará en cuatro capítulos que tratarán sobre:

- el modelo de programación de CUDA: una introducción sencilla para presentar los principales conceptos implicados en los programas en dicho modelo
- los algoritmos sobre grafos: distintos algoritmos sobre grafos implementados en CUDA, con versiones secuenciales y paralelas en distintas alternativas, junto con la evaluación de su rendimiento
- otras aplicaciones de CUDA en programación de propósito general, en las cuales facilitaremos ejemplos de problemas comunes en cada área
- un capítulo final en el que se expone una serie de conclusiones de nuestro trabajo.

2. CUDA (Compute Unified Device Architecture)

Básicamente, las GPUs implementan lo que se conoce como la tubería gráfica. Debido a que ésta es un algoritmo inherentemente paralelo en el que se llevan a cabo cómputos de forma intensiva, la evolución de las GPUs ha ido en la línea de elevar el rendimiento en la ejecución de este algoritmo, en particular, vía la programación del mismo. Por otra parte, el bajo coste de las GPUs permite encontrarlas, hoy día, en casi todos los ordenadores personales. Sin embargo, la forma en que se programaban inicialmente era muy técnica, lo que obligaba a sus potenciales programadores a conocer detalles de la tubería gráfica, así como de APIs del estilo de OpenGL o DirectX. Por este motivo se desarrollaron tecnologías como CUDA.

CUDA se basa en un modelo de programación paralela que funciona como una extensión del lenguaje C, lo cual hace que su aprendizaje no suponga dificultades importantes para los programadores familiarizados con este lenguaje.

A partir de ahora nos vamos a referir siempre a dos ámbitos dentro de un mismo sistema local:

- *Host:* que será el ordenador que albergue la tarjeta gráfica y también el que se encargará de regir su comportamiento
- *Device:* que se referirá a la tarjeta gráfica.

2.1. Modelo de programación

El componente principal del modelo de programación de CUDA son las funciones que se invocan desde el *host*, pero se ejecutan en el *device*. Dichas funciones reciben el nombre de *kernels*.

Cuando se invoca un *kernel*, éste se ejecuta N veces en N hilos (*threads)* diferentes. A cada uno de esos *threads* se les da un identificador único, que es accesible desde dentro del *kernel* a través de una variable *built-in* llamada **`threadIdx`**. Con esta variable que identifica al hilo se puede definir el comportamiento específico de cada uno de ellos.

Los *kernels* deben tener obligatoriamente tipo devuelto **`void`**, han de llevar la etiqueta **`__global__`** , se invocarán desde el *host* y se ejecutarán en el *device*.

Todos los hilos a los que dé lugar la ejecución de un *kernel* ejecutan el mismo programa (el propio *kernel*). El número de hilos debe ser conocido de antemano de manera que puedan ser tratados por lotes, agrupándolos en bloques del mismo tamaño, de acuerdo a lo especificado por el programador en la llamada al *kernel*.

2.1.1. Disposición espacial de los hilos

CUDA organiza los hilos agrupándolos en bloques de hasta tres dimensiones, que se utilizarán según la necesidad del programador.

Por este motivo la variable antes mencionada **`threadIdx`** es realmente una variable de tipo **`dim3`** que posee tres componentes `(x, y, z)`, así cada hilo podrá ser individualizado utilizando uno de estos tres índices, dando forma a un bloque de hilos (*thread block*) de una, dos o tres dimensiones.

En este trabajo se utilizarán a lo sumo dos dimensiones para los bloques, de forma que el índice `z` nunca será utilizado.

La organización en bloques permite que los hilos pertenecientes a un mismo bloque puedan cooperar entre sí compartiendo datos en lo que veremos constituye la memoria compartida (*shared memory*). También se puede coordinar su ejecución a través de la función **`__syncthreads()`**, que actúa como barrera de sincronización ante la cual los hilos de un bloque tienen que esperar a que el resto llegue para poder continuar con su ejecución.

El número máximo de hilos que puede admitir un bloque es 512. Para contrarrestar esta limitación, un *kernel* puede ser ejecutado por **múltiples bloques** de hilos de **igual forma**. De esta manera, el número total de hilos en ejecución será igual al número de bloques multiplicado por el número de hilos en cada bloque.

Los bloques pueden estar organizados en una rejilla unidimensional o bidimensional. Cada bloque de la rejilla está identificado por una variable *built-in* accesible dentro del *kernel*, llamada **`blockIdx`**. Además la dimensión de cada bloque se puede obtener mediante la variable **`blockDim`**.

La organización y tamaño de la rejilla de bloques es el primer parámetro especial que se le pasa a un kernel cuando se invoca. El segundo parámetro es el tamaño y la organización que va a tener cada bloque. Estos parámetros van encerrados dentro de un operador especial **`<<<…>>>`**.

Declaración de un kernel:

```
__global__ void sumarMatrices(float A[N][N],
                              float B[N][N],
                              float R[N][N])
{ ... }
```

Invocación de un kernel:

```
sumarMatrices<<<dimRejilla, dimBloque>>>(A, B, R);
```

Es importante destacar que la ejecución de cada bloque debe ser independiente. Tiene que ser posible ejecutar cada bloque en cualquier orden, ya sea en paralelo o en serie. Este requisito de independencia permite que cada bloque de hilos sea asignado a cualquier procesador y en cualquier orden, lo que aporta escalabilidad al código.

Generalmente el número de bloques de una rejilla viene determinado por el tamaño de los datos que se van a procesar y no por el número de procesadores de los que dispone el sistema.

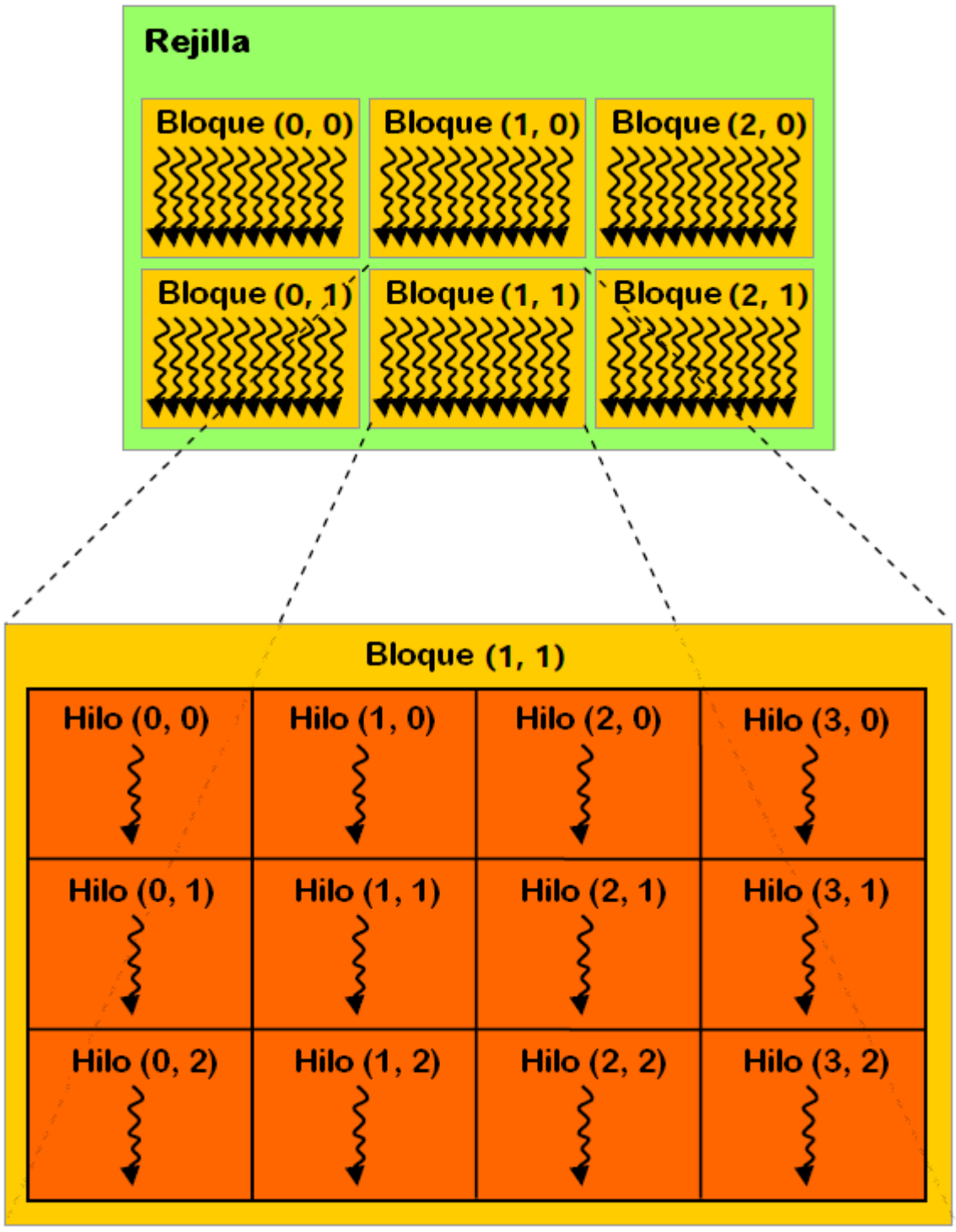


Figura 2-1. Estructura de una rejilla de bloques de hilos

2.2. Modelo de memoria

Los hilos de CUDA pueden acceder a distintos tipos de memoria durante la ejecución de un *kernel*, como se ilustra en la figura 2-2.

Cada hilo tiene acceso a una memoria local privada. Cada bloque tiene acceso a una memoria compartida a la que acceden todos los hilos del bloque y con el mismo tiempo de vida del bloque. Por último, todos los hilos tienen acceso a una memoria global.

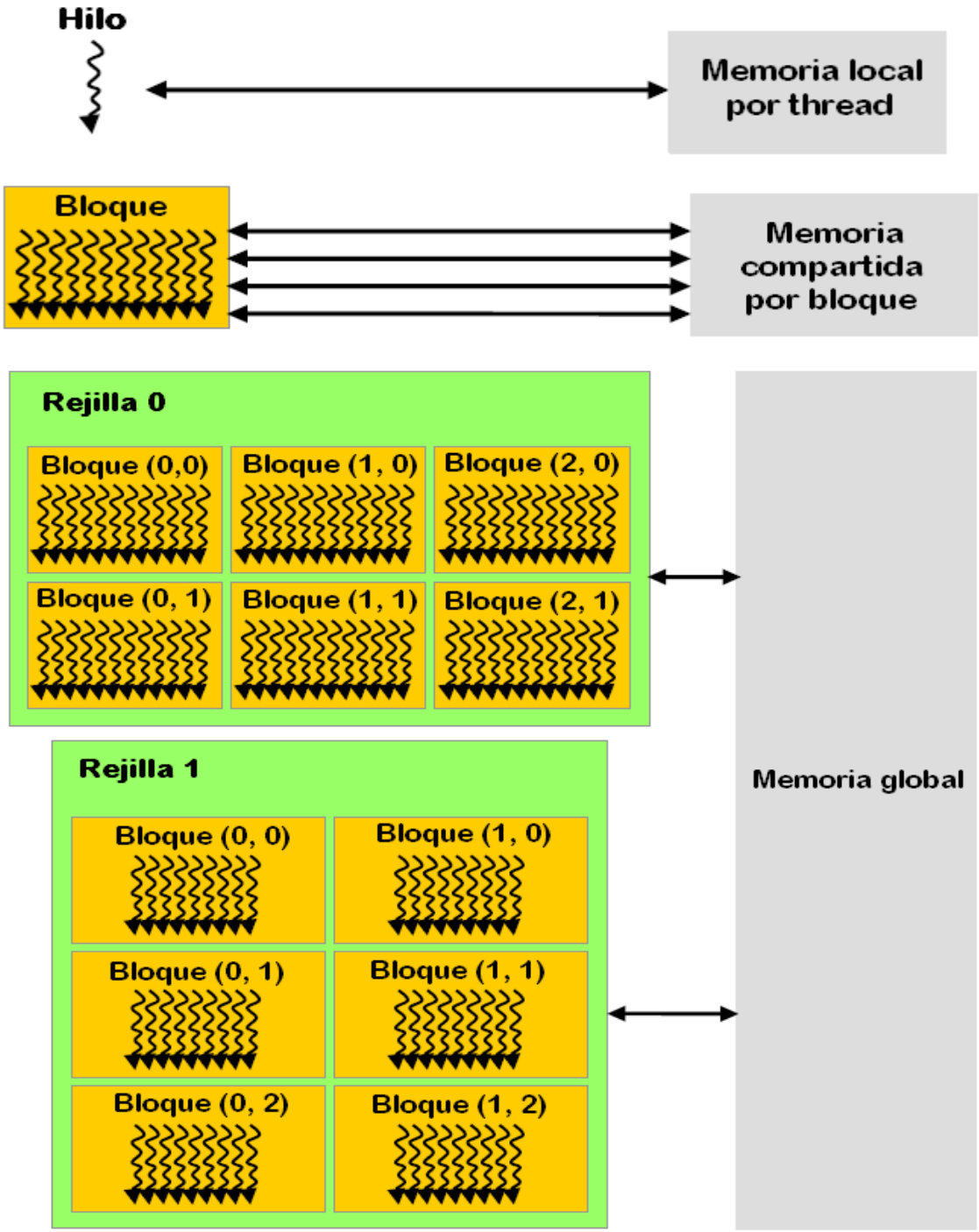


Figura 2-2. Jerarquía de memoria

Existen también dos espacios de memoria de acceso global de sólo lectura, accesibles por todos los hilos: el espacio de *constantes* y el espacio de *texturas*.

2.2.1. Memoria global

Estos tres espacios de memoria de acceso global están optimizados para diferentes usos. Asimismo, son persistentes a través de los diferentes *kernels* que ejecute una misma aplicación.

El espacio de *memoria global* no es cacheado y es capaz de leer palabras de 2, 4 u 8 bytes en una única instrucción de lectura.

El espacio de *constantes* está cacheado, el valor solicitado se lee desde la caché de constantes y sólo se lee la memoria global en un fallo de caché. Leer de la caché de constantes puede ser tan rápido como leer de un registro, siempre que todos los hilos de un *warp* (se explica en la siguiente sección) lean la misma dirección.

El espacio de *texturas*, al igual que el de constantes, está cacheado, pero con la diferencia de que está optimizado para localidad espacial 2D. De modo que los hilos pertenecientes a un mismo *warp* que lean direcciones que estén cercanas conseguirán un mejor rendimiento. Además tiene un tiempo de latencia constante, independientemente de si el dato se lee de caché o de memoria; la única diferencia es que reduce el ancho de banda utilizado con la memoria. De esta manera, resulta ventajoso leer datos del espacio de texturas en lugar de hacerlo del espacio de memoria global o de constantes.

2.2.2. Memoria compartida

Como se explicó anteriormente, se trata de una memoria común a cada bloque de hilos. Está construida *on-chip*, con lo cual un acceso a ella es mucho más rápido que uno a memoria global. De hecho, para todos los hilos de un *warp*, acceder a la memoria compartida es tan rápido como acceder a un registro, siempre y cuando no haya conflictos de bancos entre los hilos.

Para obtener un mayor ancho de banda, la memoria compartida de cada multiprocesador se encuentra dividida en módulos de memoria de igual tamaño llamados **bancos**, que pueden ser accedidos simultáneamente. De esta forma, la memoria puede atender con éxito un pedido de lectura o escritura constituido por n direcciones que pertenezcan a n bancos distintos. Así se consigue un ancho de banda n veces mayor al de un banco simple.

Ahora bien, si dos direcciones de un pedido caen en el mismo banco de memoria, hay un conflicto y el acceso tiene que ser serializado. El hardware entonces divide el acceso a memoria en tantos accesos libres de conflicto como sean necesarios, perdiendo obviamente ancho de banda.

Para conseguir el máximo rendimiento, es importante conocer la organización de los bancos de memoria, y cómo se organiza el espacio de direcciones. Así se podrán programar los accesos a memoria compartida de manera que se minimicen los conflictos.

Los bancos están organizados de forma que palabras sucesivas de 32-bits pertenecen a bancos sucesivos (distintos). El ancho de banda de cada banco es de 32-bits cada 3 ciclos de reloj. El número de bancos es de 16, para tarjetas de capacidad 1.X.

2.3. Organización de la ejecución

CUDA asume que los hilos se ejecutarán en un dispositivo físico separado (*device*) que opera como un coprocesador del *host*, que es quien se encarga de ejecutar el programa en C. Además también se asume que *host* y *device* tienen su propia memoria DRAM.

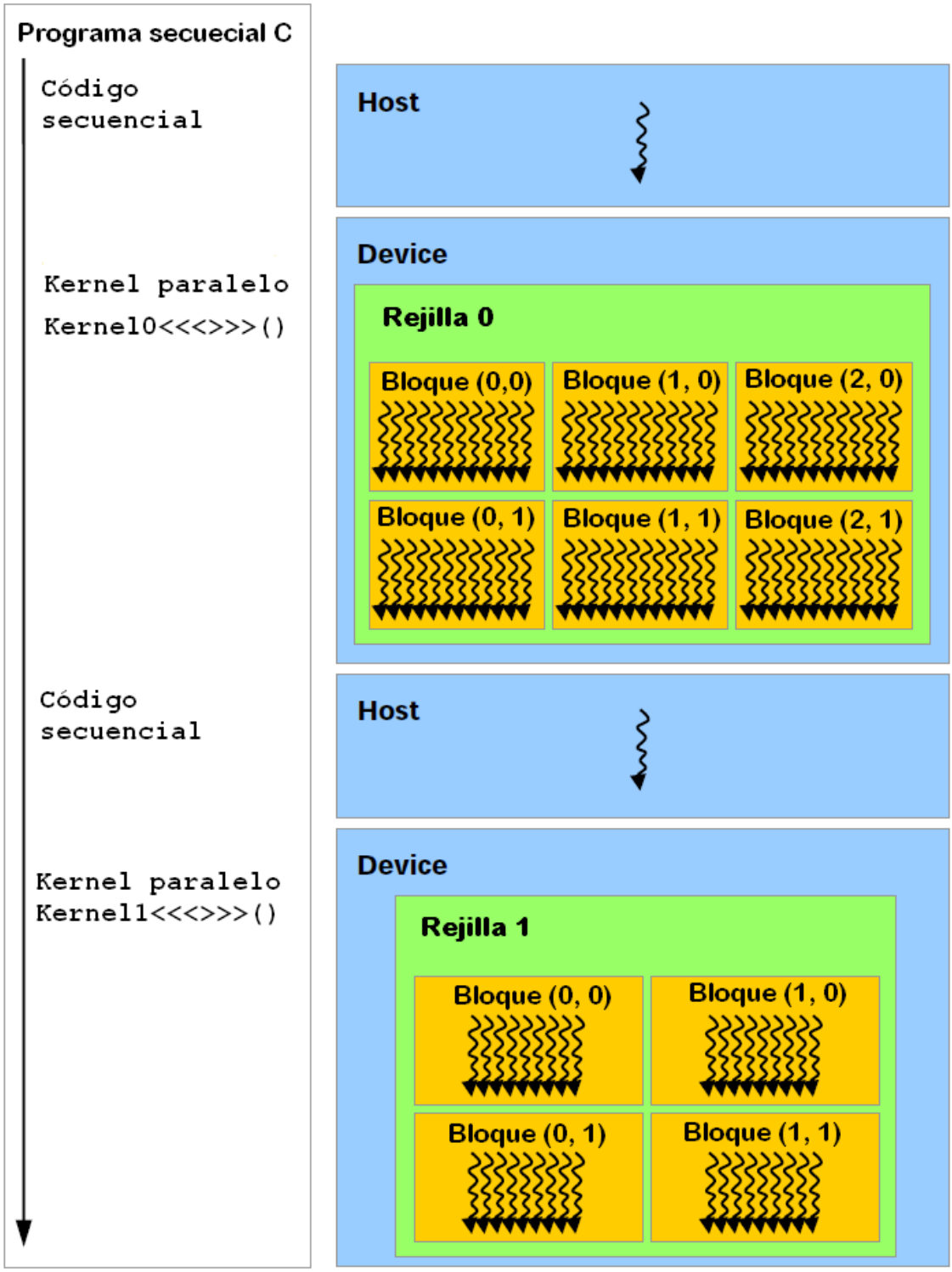


Figura 2-3. Ejecución de un programa CUDA.

El modelo de ejecución de los programas CUDA está íntimamente ligado a la arquitectura de la GPU. Esta arquitectura está basada en un array escalable de *Streaming Multiprocesors (SMs)*.

Cuando un programa CUDA invoca un *kernel* con una disposición dada, los bloques de la rejilla son enumerados, y distribuidos a los *multiprocesadores* con capacidad de ejecución disponible. Los hilos de un bloque se ejecutan concurrentemente en un único multiprocesador. Cuando un bloque termina, nuevos bloques son lanzados en los multiprocesadores vacantes.

Un multiprocesador SM consiste en:

- ocho núcleos de Procesadores Escalares (SP) con 32 registros cada uno,
- dos unidades de funciones especiales,
- una unidad de instrucciones multihilo,
- memoria compartida *on-chip*,
- una *caché de constantes* de sólo lectura compartida por todos los SP, y
- una *caché de texturas* de sólo lectura también compartida por todos los SP.

El SM crea, gestiona y ejecuta los hilos concurrentemente en hardware, sin sobrecarga de planificación. También implementa la función **`__syncthreads()`** que actúa como barrera de sincronización de una única instrucción.

Las dos características mencionadas hacen que se soporte de manera eficiente el paralelismo de grano fino, permitiendo, por ejemplo, la descomposición de problemas al asignar una unidad de datos a cada hilo (un píxel en una imagen, una celda en un sistema de cuadrículas, etcétera).

Para poder administrar cientos de hilos ejecutando diferentes programas, el multiprocesador emplea una arquitectura denominada SIMT (*single-instruction, multiple thread*). El multiprocesador asigna un hilo a cada procesador escalar y cada uno ejecuta código de forma independiente basándose en sus registros de estado. El multiprocesador SIMT crea, administra, programa (organiza la ejecución) y ejecuta los hilos en grupos de 32 hilos paralelos llamados *warps*. Un *half-warp* puede estar formado por los primeros o los segundos 16 hilos de un *warp*. Los hilos que componen un *warp* empiezan a ejecutar código de forma conjunta en la misma dirección de programa, y son libres para tomar bifurcaciones y ejecutarse independientemente.

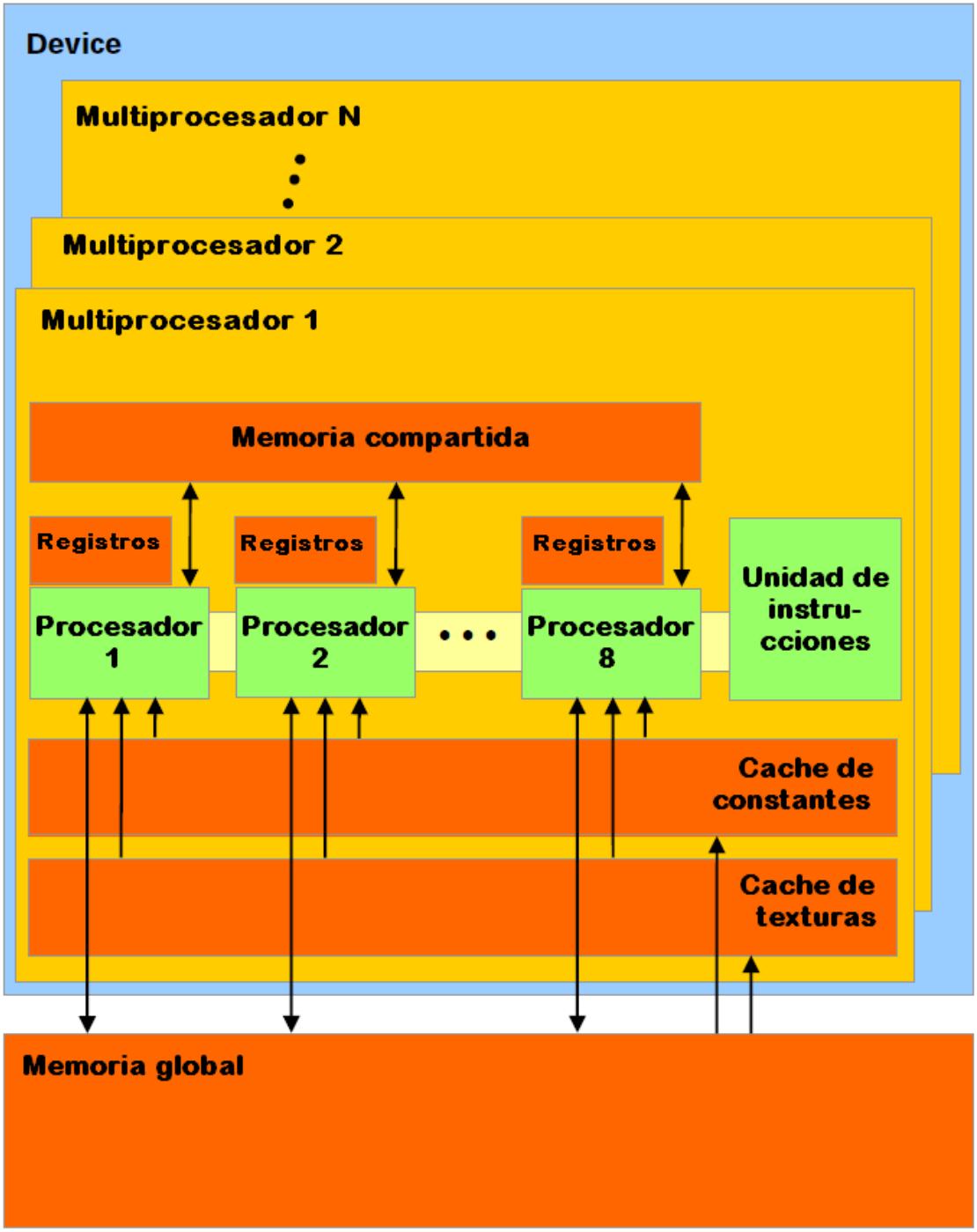


Figura 2-4. Arquitectura SIMT

La forma en que un bloque es dividido en *warps* es siempre la misma, cada *warp* contiene hilos con ID creciente, con el primer *warp* conteniendo el hilo 0.

La unidad SIMT selecciona un *warp* que esté listo para ejecutar y prepara la siguiente instrucción para los hilos activos del *warp*. Un *warp* ejecuta una instrucción común cada vez, de modo que la mayor eficiencia se obtiene cuando los 32 hilos del *warp* coinciden en su camino de ejecución, es decir, que si hay una bifurcación condicional, todos los hilos siempre toman el mismo salto. Si hay hilos que divergen, el *warp* ejecuta en forma serializada cada rama, deshabilitando los hilos que no están en esa rama; cuando todas las

ramas terminan, los hilos convergen nuevamente. La divergencia de saltos sucede únicamente entre los hilos de un *warp*; *warps* diferentes ejecutan código de forma independiente, y no interesa que estén ejecutando caminos comunes o disjuntos.

Cuántos bloques puede procesar un multiprocesador de una vez depende de cuánta memoria compartida se necesite. Si no hay suficientes registros o memoria compartida por multiprocesador para procesar como mínimo un bloque, el *kernel* fallará en su ejecución. Un SM puede ejecutar hasta 8 hilos de un bloque de forma concurrente.

2.4. API de CUDA

En esta sección vamos a exponer algunas de las funciones de uso habitual u obligado en un programa CUDA, y aquellas extensiones que hemos estado comentando a lo largo del presente capítulo.

2.4.1. Modificadores de funciones

Modificador	Significado	Restricciones
`__global__`	Función *kernel*, se invocará en el *host*, y se ejecutará en el *device*.	Tipo de retorno `void`. No puede ser recursiva. No puede declarar variables `static`. No puede tener un número variable de argumentos. Los parámetros están limitados a 256 bytes.
`__device__`	Se ejecuta en el *device* y sólo se puede llamar desde otras funciones *device* o *global*.	No puede tomarse su dirección para pasarla por parámetro a otra función. No puede ser recursiva. No puede declarar variables `static`.
`__host__`	Modificador por defecto. Estas funciones tienen todas las cualidades a las que estamos acostumbrados.	Se puede utilizar junto con el modificador `__device__`, y la función se compilará tanto para ejecutar en *host* como para hacerlo en *device*.

2.4.2. Modificadores de variables

Modificador	Significado	Restricciones
`__device__`	Declara una variable que reside en la memoria global. Tiene el tiempo de vida de la aplicación y es accesible por todos los hilos y el *host*.	Se puede utilizar junto con los otros dos modificadores.
`__constant__`	Reside en el espacio de constantes y tiene el tiempo de vida de la aplicación. Es accesible desde todos los hilos y desde el *host*.	Sólo pueden ser asignadas desde el *host*.
`__shared__`	Reside en el espacio de memoria compartida de un bloque. Tiene el tiempo de vida de un bloque y sólo es accesible por los *threads* de un bloque.	No puede tener un valor inicial en su declaración. Si se escribe, su valor está garantizado que será visible por todos los hilos, sólo después de un `__syncthreads()`.

Ninguno de los 3 tipos de modificadores se puede utilizar en variables definidas como externas con el modificador **`extern`**.

Tampoco se pueden utilizar en miembros de uniones (**`union`**) o estructuras (**`struct`**).

2.4.3. Variables built-in

Nombre	Tipo	Significado
`gridDim`	`dim3`	Contiene las dimensiones de la rejilla de bloques.
`blockDim`	`dim3`	Contiene la dimensión del bloque.
`blockIdx`	`uint3`	Contiene el índice del bloque dentro de la rejilla. La componente z nunca se usa.
`threadIdx`	`uint3`	Contiene el índice del hilo dentro del bloque.
`warpSize`	`int`	Contiene el tamaño del *warp* en número de *threads*.

Los tipos `uint3` y `dim3` son idénticos; su única diferencia es que en el tipo `dim3`, si alguna de sus componentes `(x,y,z)` no se inicializa, se pone a 1 en lugar de ser 0.

A ninguna de estas variables se les puede asignar valores, ni tampoco tomar su dirección.

2.4.4. Runtime API

CUDA posee un abanico de funciones matemáticas que pueden usarse en los programas, tanto en *host* como en *device* (*Common Runtime Component*) y otras para usar exclusivamente en *device* (*Device Runtime Component*). Todas vienen especificadas en el

apéndice B de **[3]**. Pero vamos a destacar de entre ellas, las funciones **`max(x,y)`** y **`min(x,y)`** que implican una sola instrucción en *device*, si se llaman con enteros.

CUDA también posee una serie de funciones atómicas para usar en funciones *device*, que operan sobre una posición de memoria siguiendo la secuencia: lectura del valor, operación, escritura del valor. Lo hacen como una única unidad sin que ningún otro hilo opere en medio de dicha secuencia. Dichas funciones vienen enumeradas en el apéndice C de **[3]**.

También hay que destacar las funciones de transferencia de memoria entre *host* y *device*, así como las funciones de asignación de memoria lineal, que son las que vamos a utilizar más asiduamente.

Función	Utilidad
`cudaMalloc(void** ptr, size_t size)`	Asignación de memoria lineal. Deja en `ptr` una dirección de memoria global.
`cudaFree(void* ptr)`	Libera la memoria pedida con `cudaMalloc`.
`cudaMemcpy(void* hostPtr, void* devPtr, size_t size, cudaMemcpyDeviceToHost)`	Copia memoria desde la memoria global del *device* a la memoria del *host*.
`cudaMemcpy(void* devPrt, void* hostPtr, size_t size, cudaMemcpyHostToDevice)`	Copia memoria desde el *host* al espacio de memoria global del *device*.

2.5. Ejemplo

Para comprender mejor el funcionamiento de un programa CUDA, vamos a presentar un ejemplo sencillo que consiste en la suma de dos matrices. Cada hilo se va a encargar de sumar una componente de las mismas.

Por tanto vamos a tener tantos hilos como elementos tengan las matrices, y vamos a organizar la ejecución en bloques de tamaño 16×16 hilos. Además, armaremos una rejilla bidimensional de bloques N/16×M/16, suponiendo que N y M son ambos, múltiplos de 16.

```
#define BLOCK_SIZE 16


__global__ void sumarMatrices_Kernel(const float* A,
   const float* B,
   int filas,
   float* C)
{
   int iGlobal = blockIdx.x * blockDim.x + threadIdx.x;
   int jGlobal = blockIdx.y * blockDim.y + threadIdx.y;
   int indice = jGlobal * filas + iGlobal;
   C[indice] = A[indice] + B[indice];
}

void sumarMatrices_GPU(const float* A,
   const float* B,
   int filas, int cols,
   float* C)
{
   int size = filas * cols * sizeof(float);

   // Cargar A en la memoria del device
   float* Ad;
   cudaMalloc((void**)&Ad, size);
   cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);

   // Cargar B en la memoria del device
   float* Bd;
   cudaMalloc((void**)&Bd, size);
   cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

   // Reservar espacio para C en el device
   float* Cd;
   cudaMalloc((void**)&Cd, size);

   // Crear la configuración de la ejecución asumiendo
   // que las dimensiones de la matriz son múltiplos de
   // BLOCK_SIZE
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
   dim3 dimGrid(filas / dimBlock.x, cols / dimBlock.y);

   // Invocación del kernel para que sume en paralelo
   sumarMatrices_Kernel<<<dimGrid, dimBlock>>>(Ad, Bd, filas, Cd);

   // Leer C desde device
   cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

   // Liberar la memoria del device
   cudaFree(Ad);
   cudaFree(Bd);
   cudaFree(Cd);
}
```

2.6. Resumen de conceptos

- Un kernel puede ser ejecutado por múltiples bloques de igual forma.
- Los bloques se pueden organizar en rejillas de 1 o 2 dimensiones.
- Los hilos dentro de un bloque se pueden organizar hasta en 3 dimensiones.
- Cada bloque ejecuta código de forma independiente. Se pueden ejecutar en cualquier orden, en serie o en paralelo.
- Cada hilo tiene acceso a 3 espacios de memoria: local, de bloque y global. La memoria global a su vez está dividida en 3 espacios: texturas, constantes y ordinaria.
- Los hilos de un bloque pueden compartir memoria a través de la memoria compartida, de baja latencia.
- Se puede coordinar la ejecución de los hilos con la llamada **`__syncthreads()`**.
- Todos los hilos pertenecientes a un bloque se ejecutan en el mismo multiprocesador.

3. Algoritmos sobre grafos

Hay distintos algoritmos que se han convertido en clásicos dentro de la teoría de grafos. Estos algoritmos tienen diversos objetivos como realizar búsquedas dentro de un grafo, encontrar los caminos mínimos desde un único origen, encontrar los caminos mínimos entre pares de vértices u obtener el árbol de recubrimiento mínimo de un grafo.

En este capítulo, vamos a hacer un estudio sobre la paralelización de estos algoritmos utilizando el modelo de programación de CUDA.

3.1. Motivación

El grupo de algoritmos que vamos a estudiar ha sido objeto de investigación desde los inicios de la informática, ya que resuelven problemas que se presentan en un gran número de situaciones. Hasta hace no mucho tiempo, el objetivo de la mayoría de los estudios era mejorar su rendimiento en una ejecución secuencial.

Por nuestra parte, el objetivo de nuestro estudio ha sido mejorar el rendimiento que se puede obtener con estos algoritmos secuenciales haciendo uso de un modelo de programación en paralelo. Con la paralelización de los algoritmos surgen complicaciones que deben tenerse en cuenta, y sobre los que haremos un especial hincapié.

Los problemas que vamos a tratar de paralelizar son la búsqueda de los caminos mínimos entre todos los pares de vértices de un grafo (*All Pairs Shortest Path*, abreviado por el acrónimo APSP), y la búsqueda de un árbol de recubrimiento mínimo para un grafo (*Minimum Spanning Tree*, MSP). Para este segundo caso, partiremos de una solución ya existente al problema de los caminos mínimos desde un único origen (*Single Source Shortest Path*, SSSP).

3.2. Distintas formas de representación

Existen dos estructuras de datos ampliamente utilizadas para la representación de grafos: las listas de adyacencia y las matrices de adyacencia. Aunque generalmente el diseño de los algoritmos es independiente de la estructura de datos utilizada, las diferencias existentes entre ambas hacen que varíe el tamaño de los grafos representables utilizando uno u otro método, como también varía el rendimiento.

Los algoritmos que trataremos de paralelizar en esta sección se implementarán, siempre que sea posible, utilizando las dos estructuras de datos mencionadas. Cada una de las implementaciones tendrá un rendimiento y un rango de tamaños de grafo representables diferente.

3.3. SSSP (Single Source Shortest Path)

El problema del SSSP consiste en encontrar los caminos más cortos desde un único vértice al resto de los vértices de un grafo. Para resolver este problema se han aportado diversas soluciones secuenciales, siendo las más conocidas el algoritmo de Dijkstra y el de Bellman-Ford.

También se han diseñado implementaciones en paralelo usando CUDA que resuelven este mismo problema (véase bibliografía, referencias **[1]** y **[2]**), obteniendo resultados que mejoran notablemente el rendimiento de las versiones secuenciales.

3.3.1. Punto de partida

Nuestro estudio de la paralelización en CUDA de algoritmos sobre grafos comienza con el análisis de la implementación aportada en **[1]**. Esta implementación parte del algoritmo de Dijkstra y aporta diferentes enfoques a la hora de paralelizarlo. También este análisis resulta útil para aprender las dificultades que pueden surgir cuando se trabaja con CUDA y las medidas que se pueden tomar para superarlas.

3.3.2. Versión clásica

El algoritmo de Dijkstra resuelve el problema de los caminos más cortos desde un solo origen para grafos $G = (V, E)$ dirigidos y valorados, en los que todas las aristas $(v, v') \in E$ tienen peso positivo $\omega(v, v') > 0$. Supondremos, en lo que sigue, que los vértices están numerados de 0 a $|V| - 1$ y que el vértice de origen es el 0. El algoritmo divide el conjunto de vértices en dos partes: el conjunto R de *vértices resueltos* y el conjunto U de *vértices no resueltos*, y mantiene una estimación del camino más corto $c[i]$ para cada vértice i, que coincide con la longitud del camino más corto para los vértices resueltos. Para los vértices no resueltos, $c[i]$ almacena la longitud del *camino especial* más corto de i con respecto a R, esto es, el camino más corto de entre los caminos a i que únicamente atraviesan vértices de R antes de llegar a i.

El algoritmo de Dijkstra se compone de un bucle, cada iteración del cual está formada por tres pasos: se *relajan* las estimaciones para los vértices de U utilizando el último vértice añadido a R (que denominamos *vértice frontera*), se calcula la estimación mínima de los vértices de U y se mueve uno de los vértices con estimación mínima a R, que se convierte en el nuevo vértice frontera.

Este algoritmo, independientemente de la representación del grafo que se use, se ejecuta en un tiempo $O(n^2)$.

3.3.3. Versión paralela

Para realizar una versión en paralelo del algoritmo que resuelve el problema del SSSP, se diseñó una modificación del algoritmo de Dijkstra adaptado para que trabaje con fronteras compuestas. Como acabamos de decir, el algoritmo clásico de Dijkstra añade vértices de U a R de uno en uno. Sin embargo, puede demostrarse que el algoritmo sigue siendo correcto si se añaden simultáneamente a R todos los vértices i de U cuya estimación $c[i]$ sea mínima.

Sobre el algoritmo de Dijkstra adaptado a fronteras compuestas, y con el fin de realizar una versión paralela, hacemos observar que se puede paralelizar cada uno de los pasos que forman el bucle externo del algoritmo:

1. Relajar la estimación del camino más corto para cada vértice de *U* utilizando los vértices de la frontera compuesta. Es decir, calcular $c[j] = \min\{c[j], c[f] + \omega(f, j)\}$ para cada par de vértices $j \in U$ y $f \in F$.
2. Encontrar la estimación mínima de cada vértice de *U*.
3. Actualizar el conjunto *U* de vértices no resueltos, eliminando aquellos vértices cuya estimación es igual al mínimo encontrado en el paso anterior. Estos vértices pasan a formar la nueva frontera compuesta.

Existen diversas implementaciones para cada uno de estos pasos. Cada una de ellas se podría implementar de manera secuencial con un bucle sencillo (o dos bucles anidados para el caso de la relajación). Sin embargo, es posible una implementación que ejecute cada una de las iteraciones del bucle (del bucle externo, en el caso de la relajación) en paralelo.

La paralelización del paso de relajación admite dos enfoques diferentes. Uno de los enfoques posibles es lanzar un hilo de ejecución en paralelo para cada uno de los vértices de la frontera compuesta. Para cada uno de estos vértices, visitamos todos sus sucesores, relajando la estimación $c[i]$ para todos aquellos sucesores *i* que pertenezcan al conjunto *U*. Este enfoque, sin embargo, tiene el problema de que pueden producirse inconsistencias si varios de los hilos que se ejecutan en paralelo tratan de relajar concurrentemente la estimación para el mismo vértice. Afortunadamente, las tarjetas gráficas de NVIDIA con capacidad de cómputo igual o superior a 1.1, proporcionan operaciones atómicas que permiten evitar dichos problemas de concurrencia.

Existe otro enfoque para la relajación para el que no son necesarias estas operaciones atómicas pues no pueden darse escrituras concurrentes. Este segundo enfoque se centra en los vértices de *U* en lugar de los vértices de la frontera compuesta. En este caso, para cada vértice *i* de *U*, visitamos todos sus predecesores, relajando la estimación $c[i]$ cuando el predecesor pertenece a la frontera compuesta.

Paralelizar el cálculo del mínimo es una tarea más complicada, ya que es un cómputo inherentemente secuencial. Afortunadamente, existen técnicas de reducción en el SDK de CUDA que, adaptadas al caso concreto del mínimo, hacen posible el cálculo en paralelo del mínimo de una serie de elementos.

Por último es posible trivialmente coleccionar los vértices no resueltos *i* con estimación $c[i]$ mínima, de forma paralela sin más que preguntar por la estimación de cada elemento de *U* simultáneamente.

3.3.4. Pseudocódigo

Las ideas del algoritmo de Dijkstra adaptado a fronteras compuestas se traducen en el código mostrado en este apartado. El programa consta, como se dijo, de un bucle externo en el que se realizan tres pasos: relajación, minimización (o búsqueda del mínimo) y actualización. La implementación viene precedida de una inicialización de las variables que se utilizan en el algoritmo.

```
void SSSP(c) {
  inicializa(c, f, u);
  min = 0;
  mientras (min != INFINITO) {
    relaja(c, f, u);
    min = minimo(c, u);
    actualiza(c, f, u, min);
  }
}
```

La variable `c` es un vector que almacena la estimación del camino más corto que va desde el vértice origen (recordemos que el vértice origen es siempre el 0) a cada uno de los demás vértices del grafo. Las variables `f` y `u` son vectores que almacenan valores booleanos indicando, respectivamente, si un vértice pertenece a la frontera y si pertenece al conjunto de vértices no resueltos.

El método `inicializa` asigna los valores iniciales a las variables del programa. La estimación inicial para cualquier vértice (a excepción del vértice origen) es la peor posible, es decir, infinito. Inicialmente, la frontera la forma el vértice origen y el resto de los vértices son vértices no resueltos.

```
void inicializa(c, f, u) {
  para cada vértice i {
    c[i] = INFINITO;
    f[i] = falso;
    u[i] = cierto;
  }
  c[0] = 0;
  f[0] = cierto;
  u[0] = falso;
}
```

El primero de los enfoques de paralelización del método de relajación recorre todos los vértices y, para cada vértice *i* de la frontera, relaja la estimación de cada uno de sus sucesores *j*. Es decir, si existe un *camino especial* a *i* pasando por *j* más corto que el estimado hasta ahora, se sustituye la estimación por el coste de dicho camino especial. Esto se corresponde con el siguiente código:

```
void relaja(c, f, u) {
  para cada i en paralelo {
    si(f[i]) {
      para cada j sucesor de i {
        si(u[j]) {
          atomicMin(c[j],c[i]+w[i,j]);
        }
      }
    }
  }
}
```

El procedimiento `atomicMin` es una de las operaciones atómicas que forman parte del API de CUDA. Este procedimiento recibe dos argumentos, calcula el mínimo de ambos y almacena el resultado en el primero de ellos.

El segundo enfoque para paralelizar la relajación de las estimaciones recorre el conjunto de vértices no resueltos *U*, y sustituye la estimación actual por una de menor coste en caso de que exista. La implementación de esto produciría un código como el siguiente:

```
void relaja(c, f, u) {
  para cada i en paralelo {
    si(u[i]) {
      para cada j predecesor de i {
        si(f[j]) {
          c[i] = min(c[i],c[j]+w[j,i]);
        }
      }
    }
  }
}
```

El paso de actualización también puede paralelizarse de manera sencilla, obteniendo el código que se muestra a continuación:

```
void actualiza(c, f, u, min) {
  para cada i en paralelo {
    f[i] = falso;
    si(c[i] == min) {
      u[i] = falso;
      f[i] = cierto;
    }
  }
}
```

La actualización toma los vértices de *U* cuya estimación sea mínima y los mueve a *R*. Estos mismos vértices pasan a formar también la nueva frontera.

3.3.5. Implementaciones realizadas

Para cada una de las dos estructuras de datos que utilizamos para la representación de grafos se realizaron varias implementaciones con el fin de encontrar la que proporcionará un mejor rendimiento.

Para las matrices de adyacencia se implementaron en CUDA cinco versiones diferentes del algoritmo de Dijkstra adaptado a fronteras compuestas. Recordemos que se podían adoptar dos enfoques diferentes en la relajación.

Las dos primeras implementaciones se basan en el primero de los enfoques mencionados. La implementación F^{GPU} realiza, en paralelo, para cada vértice *i* de la frontera compuesta, la relajación de todos los vértices a los que se puede llegar desde *i*. Esta implementación, sin embargo, no hace uso de las operaciones atómicas mencionadas anteriormente, por lo que es incorrecta y sólo tiene utilidad para comparar el rendimiento con la siguiente implementación. La segunda implementación, llamada F^{GPU_Atomic}, sí hace uso de las operaciones atómicas y es, por tanto, una solución correcta.

Las otras tres implementaciones se basan en el segundo enfoque para la relajación. Además de la implementación U^{GPU}, que plasma directamente el algoritmo cuyo pseudocódigo se mostró en el anterior apartado, existen otras dos implementaciones (U^{GPU_T} y U^{GPU_RT}) que tratan de obtener mejor rendimiento haciendo uso de la memoria de texturas y la memoria compartida. El uso de la memoria de texturas permite aprovechar la

memoria *caché* y reducir los tiempos de acceso a los datos. Algo similar ocurre con la memoria compartida que, pese a su reducido tamaño, proporciona tiempos de acceso muy pequeños.

Sin embargo, las matrices de adyacencia ocupan un espacio en memoria del orden de $O(n^2)$, lo cual limita mucho el número de vértices que puede tener el grafo a analizar. Las listas de adyacencia, en cambio, requieren menos espacio en memoria, del orden de $O(n)$. Este hecho hace que las implementaciones que utilizan esta estructura de datos sean mucho más interesantes, puesto que permiten analizar grafos mucho más grandes.

Existen un total de 15 implementaciones diferentes que utilizan listas de adyacencia. Las tres primeras son implementaciones híbridas, que se ejecutan parte en la GPU y parte en la CPU. Todas ellas utilizan el segundo enfoque de los mencionados para la relajación. La implementación H1 ejecuta en GPU solamente el paso de actualización. En la H2, se ejecuta solamente la relajación en GPU. En la última implementación híbrida, la H3, se ejecutan en GPU ambos pasos (actualización y relajación), aunque la minimización se sigue realizando en CPU.

El paso que hasta ahora se ha realizado en CPU, la minimización, es paralelizable, acelerando el proceso de ejecución. Para ello, se utiliza la técnica de reducción. Esta técnica consiste en reducir el número de mínimos a la mitad, en cada iteración. La implementación U^{GPU} realiza una etapa de minimización en GPU, pero, cuando el número de mínimos es pequeño, el mínimo de esos mínimos se calcula en CPU. Otra implementación, llamada $U^{GPU+2MIN}$ realiza dos etapas de minimización en GPU, reduciendo así los cálculos que se dan en la CPU. Otras seis implementaciones son modificaciones de $U^{GPU+2MIN}$ que tratan de mejorar su rendimiento utilizando memoria de texturas y memoria compartida.

Hay dos implementaciones que utilizan un recorrido de los vértices de la frontera para relajar los costes estimados. Estas implementaciones se llaman F^{GPU} y F^{GPU_Atomic}. La primera de ellas es, como en el caso de las matrices de adyacencia, una implementación incorrecta utilizada solo para comparar su rendimiento.

3.3.6. Medidas de rendimiento

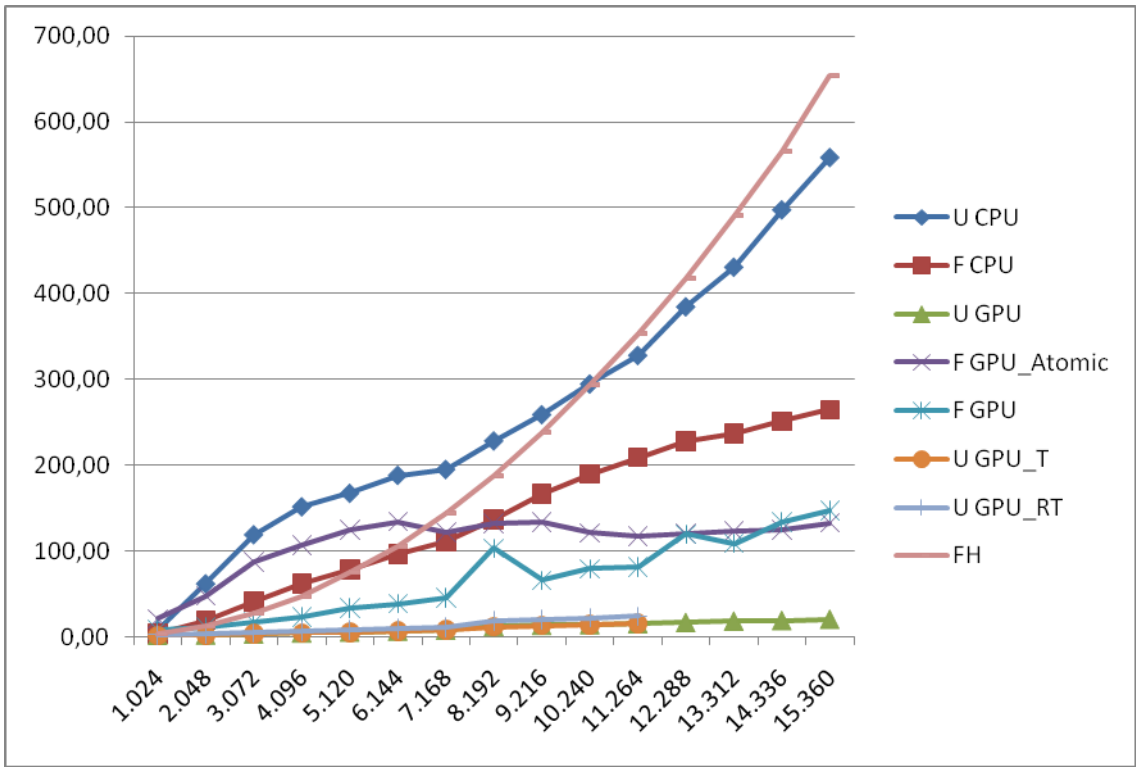
La siguiente tabla muestra el tiempo medio, en milisegundos, invertido en la ejecución de cada una de las implementaciones que utilizan matrices de adyacencia. Además de las implementaciones comentadas en el apartado anterior, se incluye una implementación que hace uso de Fibonacci-Heaps (FH). Esta implementación se incluye para comparar el rendimiento pues es, teóricamente, la implementación con mejor tiempo asintótico.

Los tiempos mostrados se calculan como la media aritmética de las 25 ejecuciones realizadas para cada tamaño de grafo.

Nº vért.	U^{CPU}	F^{CPU}	U^{GPU}	F^{GPU_At}	F^{GPU}	U^{GPU_T}	U^{GPU_RT}	FH
1024	7,99	3,63	1,53	21,01	8,09	1,53	2,05	3,11
2048	61,91	19,18	2,16	47,51	10,93	2,16	3,09	12,14
3072	119,06	40,97	3,17	87,32	16,76	3,17	4,55	27,06
4096	151,56	61,53	4,29	106,39	23,24	4,30	6,12	47,58
5120	167,25	78,24	5,42	124,88	33,29	5,40	7,72	75,33
6144	188,14	95,81	6,53	134,07	38,47	6,51	9,23	105,88
7168	194,96	111,18	7,63	121,45	45,25	7,60	10,76	144,04
8192	228,19	136,38	12,04	132,32	103,18	11,71	18,30	187,40
9216	258,74	166,59	13,29	133,25	65,69	12,95	20,30	238,10
10240	294,46	189,65	14,28	121,14	79,30	14,13	21,92	294,22
11264	327,70	208,88	15,44	117,39	81,51	15,40	23,73	353,95
12288	384,29	227,71	16,63	119,97	119,45	(*)	(*)	417,43
13312	430,45	236,79	18,25	123,33	108,78	(*)	(*)	490,45
14336	497,13	251,10	19,08	124,72	133,20	(*)	(*)	565,59
15360	558,04	264,62	20,39	132,60	147,01	(*)	(*)	653,27

(*) Con un número de vértices igual o mayor a 12288, las implementaciones U^{GPU_T} y U^{GPU_RT} dejan de funcionar correctamente, por lo que no se incluyen los tiempos invertidos en su ejecución.

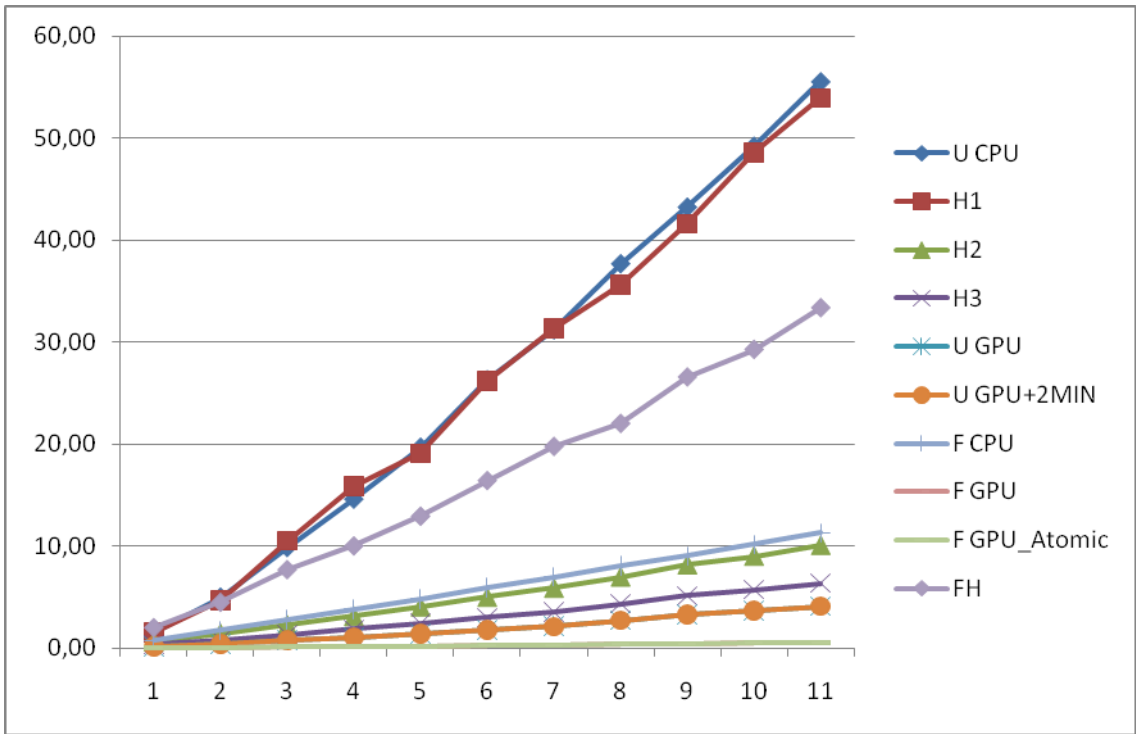
Los resultados de la tabla anterior se pueden condensar en una gráfica que permite apreciar mejor las diferencias entre los tiempos de ejecución de todas las implementaciones.



Para la implementación que hace uso de listas de adyacencia, se muestra la siguiente tabla de tiempos. En este caso, la columna de la izquierda indica el número de millones de vértices del grafo analizado. Los tiempos están indicados en segundos.

Mill. vért.	U^{CPU}	H1	H2	H3	U^{GPU}	U^{GPU}_{+2MIN}	F^{CPU}	F^{GPU}	F^{GPU_At}	FH
1	1,59	1,56	0,65	0,36	0,17	0,17	0,79	0,04	0,05	2,04
2	4,97	4,72	1,47	0,84	0,45	0,45	1,76	0,09	0,09	4,52
3	9,83	10,52	2,31	1,35	0,76	0,76	2,79	0,13	0,13	7,72
4	14,59	15,90	3,16	1,89	1,09	1,09	3,83	0,17	0,18	10,04
5	19,69	19,11	4,08	2,44	1,42	1,43	4,85	0,22	0,23	12,94
6	26,31	26,23	5,01	3,00	1,77	1,77	5,94	0,27	0,27	16,43
7	31,21	31,38	5,91	3,58	2,14	2,14	7,01	0,32	0,32	19,78
8	37,70	35,61	7,00	4,37	2,74	2,74	8,06	0,39	0,38	22,03
9	43,25	41,62	8,19	5,15	3,29	3,29	9,04	0,45	0,45	26,57
10	49,24	48,58	9,02	5,74	3,69	3,69	10,19	0,52	0,52	29,27
11	55,56	53,94	10,09	6,38	4,12	4,12	11,30	0,59	0,59	33,35

La siguiente gráfica condensa la información de la tabla anterior.



En ambos casos se puede ver cómo las implementaciones realizadas en GPU reducen considerablemente el tiempo invertido en procesar grafos de tamaño considerable.

3.4. MSP (Minimum Spanning Tree)

El problema del MSP ó del ARM (Árbol de Recubrimiento Mínimo) consiste en, dado un grafo $G=(V,E)$ conexo y valorado, encontrar un subconjunto E' de las aristas de G tal que utilizando solamente aristas de E' todos los vértices queden conectados y además la suma de los pesos de las aristas de E' cumpliendo esta condición sea mínima.

Tradicionalmente, se han utilizado dos algoritmos para resolver este problema: el algoritmo de Prim y el algoritmo de Kruskal. Cada uno de estos algoritmos adopta una estrategia diferente. El algoritmo de Prim construye, a partir de un vértice inicial, el MSP seleccionando, en cada etapa, la arista más corta que extienda el árbol. Kruskal considera en cada paso la arista más corta que conecta dos partes no conexas del árbol, hasta que todos los vértices están conectados.

Por ser su paralelización más sencilla, adoptamos el algoritmo de Prim como punto de partida para elaborar un algoritmo paralelo que resuelva el problema del MSP.

3.4.1. Versión clásica

El algoritmo de Prim resuelve el problema del árbol de recubrimiento de coste mínimo para grafos $G=(V,E)$ no dirigidos, valorados y conexos. Como en la sección anterior, suponemos que la numeración de los vértices está comprendida entre 0 y $|V|-1$. Prim parte de un vértice origen, que podemos asumir que es 0. La elección del vértice puede hacer que el árbol generado sea diferente pero, en cualquier caso, el coste es mínimo. Es decir, que sea cual sea el vértice de partida, el coste del árbol es el mismo. Al igual que el algoritmo de Dijkstra, el algoritmo de Prim divide el conjunto de vértices en un subconjunto de vértices resueltos *R* y un subconjunto de vértices no resueltos *U*. Llamamos *prometedor* a un árbol si es posible extenderlo para obtener una solución óptima. Para cada vértice *i* de *U* mantenemos el peso $e[i]$ de la arista más corta que sale de un vértice de *R* a *i*. Si $i \in R$, entonces $e[i]$ almacena el peso de la arista del *árbol prometedor* que une *i* con algún otro vértice de *R*.

El algoritmo consta de un bucle, cuyo cuerpo se puede dividir en tres pasos: se *relajan* los pesos $e[i]$ para los vértices *i* de *U* utilizando el último vértice añadido a *R* (el *vértice frontera*), se calcula el peso mínimo de los obtenidos en el paso anterior, y se mueve a *R* uno de los vértices de *U* para el que exista una arista que lo conecte con un vértice de *R* de coste mínimo, convirtiéndose en el nuevo vértice frontera.

Este algoritmo se ejecuta en un tiempo $O(n^2)$.

3.4.2. Versión paralela: aproximación desde SSSP

La forma en que se ha explicado el algoritmo de Prim recuerda bastante a la explicación del funcionamiento de Dijkstra que aparece en la solución al problema de los SSSP. Esta similitud lleva a plantear la paralelización de Prim de una manera muy similar a la que se hizo con la de Dijkstra. Sin embargo, las modificaciones realizadas en el algoritmo de Dijkstra adaptado a fronteras compuestas no son aplicables al algoritmo de Prim. Es decir, si varios vértices de *U* se pueden conectar con un vértice de *R* a través de una arista de

peso $e[i]$ mínimo, añadir estos vértices simultáneamente a *R* produce soluciones incorrectas, como podemos ver en la siguiente figura:

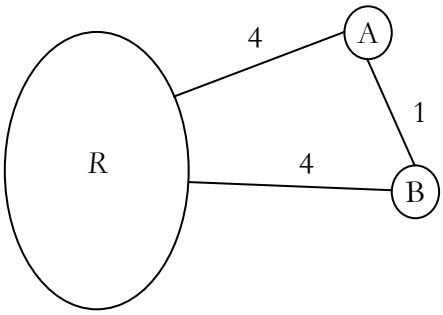


Figura 3.4.2-1. Situación en la que añadir A y B al MSP produce una solución incorrecta

En este caso, los vértices A y B pertenecientes a *U* están conectados con algún vértice de *R* mediante una arista con un peso de 4, y no hay ningún otro vértice de *U* conectado con otro de *R* mediante una arista de menor peso. Si extrapolamos el concepto de frontera compuesta aplicado al algoritmo de Dijkstra, podríamos añadir simultáneamente A y B al conjunto de vértices resueltos *R*. Sin embargo, cuando añadimos (por ejemplo) el vértice A a *R*, el peso mínimo $e[B]$ pasa a ser 1, y el árbol resultante tiene menor coste.

Existen dos alternativas que se pueden tomar a partir de este punto. La primera de ellas es prescindir de la frontera múltiple y trabajar con una única frontera como hace Prim. La otra es buscar condiciones bajo las cuales sea posible añadir varios vértices a la frontera simultáneamente.

Vamos a centrarnos, de momento, en la primera. Partimos de los tres pasos que se repiten en cada iteración del algoritmo de Prim:

1. Relajar el peso mínimo de las aristas que unen cada vértice *i* de *U* con un vértice de *R*. Esta relajación consiste en calcular, para cada vértice $i \in U$, $e[i] = \min\{e[i], \omega(f,i)\}$, donde *f* es el vértice frontera.
2. Encontrar el peso mínimo de cada vértice de *U*.
3. Actualizar el conjunto *U* de vértices sin procesar, eliminando uno de los vértices cuyo peso mínimo $e[i]$ coincida con el mínimo calculado en el paso anterior. Este vértice pasa a formar la nueva frontera.

Los dos primeros pasos se pueden implementar secuencialmente con un bucle sencillo. Para el último de los pasos, al tener una frontera compuesta por un solo elemento, la implementación es trivial si la minimización nos indica cuál de los vértices de *U* es el que va a formar la nueva frontera.

Al implementar el algoritmo de Dijkstra adaptado a fronteras compuestas en paralelo, teníamos distintos enfoques para el paso de relajación. En este caso, el primero de los enfoques (que consistiría en recorrer los vértices de la frontera actualizando los vértices de *U* conectados a éstos) no es paralelizable, porque la frontera es única. Nos queda, entonces, un único enfoque para paralelizar el paso de relajación, que es recorrer, en paralelo, cada uno de los vértices de *U* y actualizar los pesos mínimos.

En el caso de la minimización, utilizamos también la misma técnica de reducción, que esta vez devuelve no sólo el mínimo $e[i]$, sino también el vértice *i* asociado a ese mínimo.

De esta forma, el tercer paso, el de actualización, pasa a tener coste constante y no es necesario paralelizarlo.

3.4.3. Pseudocódigo

Con las indicaciones del apartado anterior, es fácil llegar a una implementación como la que describimos en esta sección. El programa consta de un bucle, y en cada iteración del mismo se realizan los tres pasos mencionados. Previamente, se inicializan las variables del programa.

```
void MST(c) {
  inicializa(e, padre, frontera, u);
  min = 0;
  mientras (min != INFINITO) {
    relaja(e, padre, frontera, u);
    (frontera, min) = minimo(e, u);
    actualiza(e, padre, frontera, u, min);
  }
}
```

El método `inicializa` asigna los valores iniciales a las variables del programa. El significado de las variables es el mismo que en la inicialización de Dijkstra, aunque hay dos diferencias fundamentales. La variable `frontera` ya no es un vector, sino que es un entero que almacena el único vértice que forma la frontera. Además, en el problema del MSP, no solamente nos interesa el coste del árbol de recubrimiento mínimo, sino también su estructura. La estructura del árbol se almacena en un vector `padre`, en el que cada posición *i* del vector almacena el padre del vértice *i*, esto es, el vértice que permitió añadir *i* a *R*. Consideramos, como raíz del árbol, aquél vértice que sea su propio padre (en este caso, siempre es 0).

```
void inicializa(e, padre, frontera, u) {
  para cada vértice i {
    e[i] = INFINITO;
    padre[i] = INFINITO;
    u[i] = cierto;

  }
  e[0] = 0;
  padre[0] = 0;
  frontera = 0;
  u[0] = falso;
}
```

La relajación de los pesos es un cálculo muy sencillo. Además de los pesos, es necesario actualizar el árbol para que se corresponda con los vértices elegidos.

```
void relaja(e, padre, frontera, u) {
  para cada i en paralelo {
    si(u[i]) {
      si(c[i] > w[frontera,i]) {
        c[i] = w[frontera,i];
        padre[i] = frontera;
      }
    }
  }
}
```

3.4.4. Implementaciones realizadas

Se han realizado implementaciones para cada una de las dos formas de representación del grafo: listas de adyacencia y matrices de adyacencia.

Para la versión de matrices de adyacencia se han realizado, en primer lugar, dos implementaciones del algoritmo en CPU para tener una referencia de tiempos y para comprobar que los resultados obtenidos en GPU son correctos. Las dos implementaciones son muy similares y únicamente tienen pequeñas diferencias orientadas a mejorar el rendimiento. La principal diferencia entre las dos implementaciones en CPU es el recorrido que se hace de los vértices en la relajación. Las implementaciones en GPU también son muy similares entre sí. La implementación GPU1 implementa el algoritmo paralelo explicado en las secciones anteriores. La implementación GPU2 implementa el mismo algoritmo haciendo uso de las variables de textura para mejorar el rendimiento.

Para la versión de listas de adyacencia existe un mayor número de posibles implementaciones. Una de las implementaciones en CPU utiliza el algoritmo de Prim, que va a ser utilizada como referencia. La otra implementación en CPU es una codificación del algoritmo de la sección 3.4.3. Existe una implementación híbrida H1, en la que solamente la relajación se realiza en GPU. Después, existen dos implementaciones, GPU1 y GPU2 que introducen la minimización en GPU utilizando técnicas de reducción en una y dos etapas, respectivamente. El resto de implementaciones son modificaciones de GPU2 que tratan de mejorar el rendimiento haciendo uso de la memoria compartida y la memoria de texturas. GPU2_S hace uso de la memoria compartida para almacenar los datos de uso frecuente en una memoria de menor latencia. GPU2_ST1, GPU2_ST2 y GPU2_ST3 hacen uso de la memoria de texturas para reducir el tiempo de acceso a los datos.

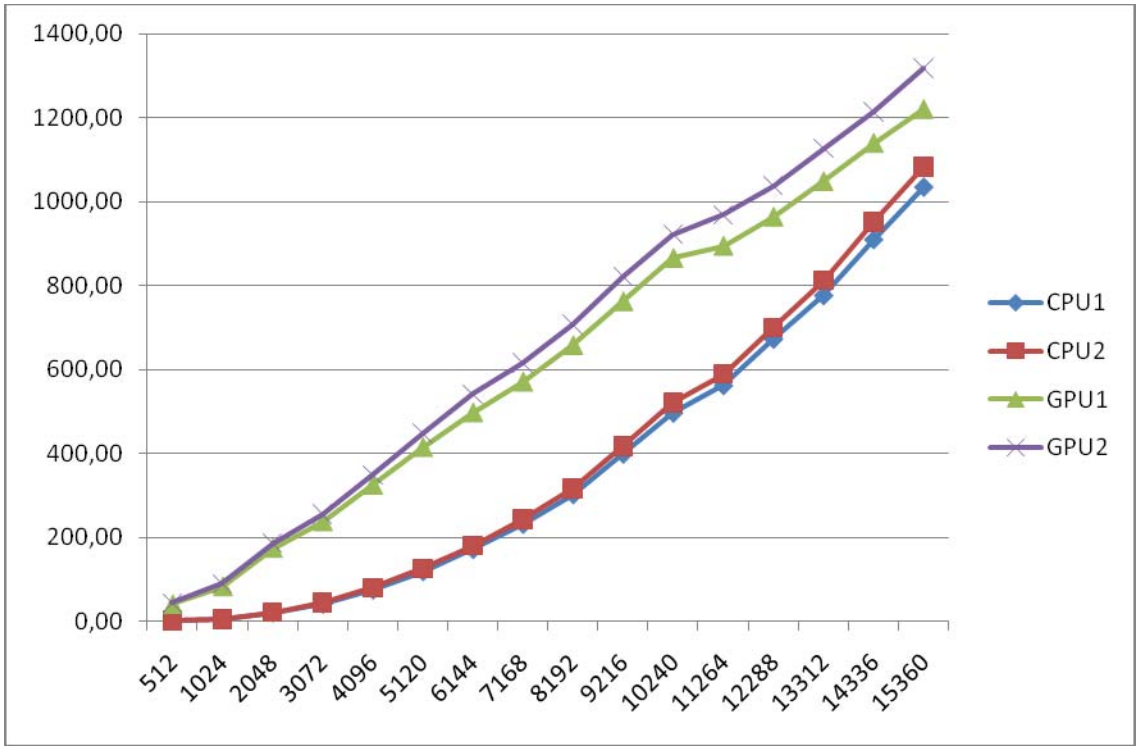
3.4.5. Medidas de rendimiento

En primer lugar vamos a analizar el rendimiento de la implementación que hace uso de una matriz de adyacencia como estructura para la representación del grafo. La siguiente tabla muestra los tiempos medios (medidos en milisegundos) que tarda cada implementación del algoritmo en completar la ejecución para distintos tamaños de grafo (en número de vértices). Los tiempos medios se obtienen calculando la media aritmética de diez ejecuciones para diez grafos diferentes.

Los grafos utilizados para estas pruebas son grafos generados aleatoriamente. Todas las aristas de los grafos tienen pesos comprendidos entre 0 y 100. El algoritmo de generación trata también de que todos los vértices tengan el mismo grado, es decir, el mismo número de conexiones con otros vértices. En este caso, el grado que se trata de obtener en todos los vértices es 6. El algoritmo cuyo rendimiento estamos analizando es sensible a la topología del grafo, por lo que es importante detallar estos parámetros de generación.

Nº vértices	CPU1	CPU2	GPU1	GPU2
512	1,27	1,31	40,02	43,43
1024	4,94	5,11	82,38	89,00
2048	20,52	21,36	174,03	186,51
3072	41,35	43,71	237,66	255,59
4096	75,18	79,73	324,51	348,60
5120	119,02	125,56	414,14	446,76
6144	171,64	180,20	497,02	541,13
7168	231,68	243,23	570,48	615,11
8192	302,27	316,67	657,14	707,88
9216	399,05	417,26	761,85	821,23
10240	496,67	520,06	864,95	922,25
11264	562,33	589,01	894,08	968,50
12288	672,93	699,92	963,19	1038,16
13312	776,77	812,19	1047,84	1126,99
14336	910,21	951,46	1139,08	1214,69
15360	1035,42	1082,27	1219,76	1318,54

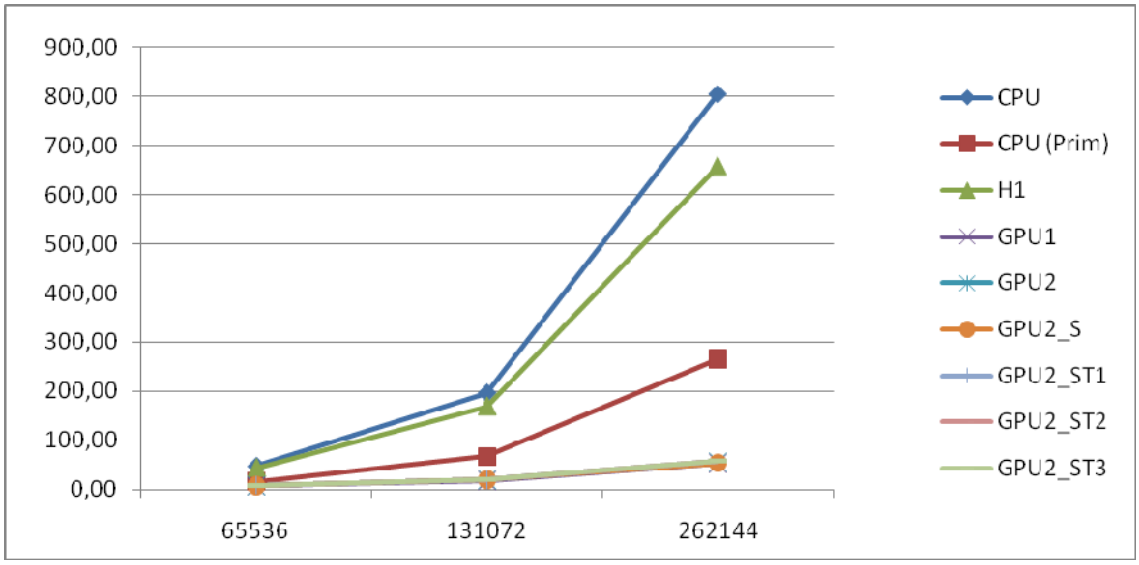
Se puede comprobar que, con la alternativa utilizada, el tiempo de proceso es menor en los algoritmos secuenciales ejecutados en CPU que en los algoritmos ejecutados en GPU en paralelo. Al representar estos resultados en forma de gráfica, podemos pensar que, para tamaños de grafo mayores, la GPU puede llegar a superar a la CPU en algún punto. Desgraciadamente, al representar el grafo como una matriz de adyacencia, no caben en memoria de la GPU grafos más grandes que los que se han probado, por lo que no podemos demostrarlo empíricamente.



Las implementaciones basadas en listas de adyacencia permiten trabajar con grafos con un número mucho mayor de vértices. Este hecho facilita la paralelización y permite obtener una mejora notable en el rendimiento respecto a la ejecución secuencial. La siguiente tabla muestra el tiempo (en segundos) empleado en la ejecución del algoritmo para cada una de sus implementaciones y tamaños del grafo.

Nº vértices	CPU	CPU (Prim)	H1	GPU1	GPU2	GPU2 S	GPU2 ST1	GPU2 ST2	GPU2 ST3
65536	47,83	16,47	44,28	7,19	8,41	8,17	8,75	8,92	8,96
131072	196,76	66,87	169,32	18,32	20,58	19,90	20,07	21,43	21,58
262144	805,21	265,30	658,67	52,42	55,93	54,18	57,32	57,69	57,59

En esta ocasión sí se obtiene una ganancia en el rendimiento, y se puede observar que la tendencia es que dicha ganancia sea mayor cuanto mayor sea el tamaño del grafo. Este hecho es más apreciable si representamos esta tabla en forma de gráfica, como la que se muestra a continuación.



También se puede observar que las modificaciones introducidas en las diferentes implementaciones GPU apenas tienen impacto sobre el rendimiento.

3.4.6. Otra alternativa: Prim adaptado a fronteras compuestas

La alternativa estudiada hasta ahora, manteniendo un único vértice en la frontera, no mejora los tiempos obtenidos al ejecutar el algoritmo de Prim en CPU salvo para grafos muy grandes que no son representables si utilizamos como estructura de datos una matriz de adyacencia. En **[4]** se introduce una mejora al algoritmo de Prim con el objetivo de procesar más de un vértice por iteración. Esta mejora implica manejar, de nuevo, el concepto de *frontera compuesta*. El enfoque adoptado en **[4]**, sin embargo, no resulta adecuado para el modelo de programación de CUDA. Sin embargo, es posible adaptar las ideas básicas planteadas en el citado artículo para diseñar el algoritmo de Prim adaptado a fronteras compuestas.

Al igual que el algoritmo de Prim clásico, el algoritmo de Prim adaptado a fronteras compuestas divide el conjunto de vértices del grafo *G* a procesar en dos subconjuntos: el conjunto de *vértices resueltos* R, y el conjunto de *vértices no resueltos U*. Inicialmente, *R* contiene únicamente un vértice. Si numeramos los vértices de *G* de 0 a $n-1$, podemos asumir que inicialmente $R = \{0\}$. Es necesario mantener también, para cada vértice *i* de *U*, el peso $e[i]$ de la arista más corta que conecta *i* con un vértice de *R*. Al trabajar ahora con una *frontera compuesta*, es necesario que, al relajar estos pesos, tengamos en cuenta todos los vértices *f* que pertenezcan a dicha frontera.

Al vértice *i* de *U* cuyo valor $e[i]$ sea mínimo, lo llamaremos *vértice favorito*. Si existen varios vértices que cumplan la anterior condición, el vértice favorito será uno cualquiera de ellos. También vamos a introducir el concepto de *vértice candidato*. Un vértice *i* es *candidato* si pertenece a *U*, y el peso $e[i]$ de la arista más corta que conecta *i* con un vértice de *R* es menor o igual que el peso de cualquier arista que conecte *i* con otro vértice de *U*.

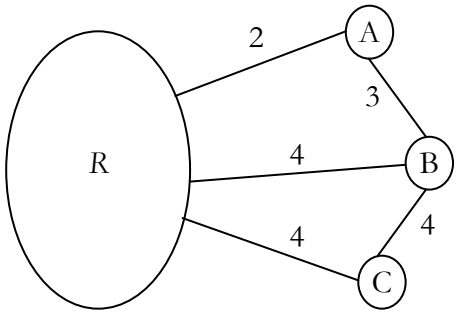


Figura 3.4.2-2. Vértices candidatos y vértice favorito

En la figura superior, los vértices A, B, y C pertenecen a *U*. A es el vértice favorito, ya que es el vértice más cercano a *R*. El vértice A es además un vértice candidato, ya que su distancia a *R* es menor o igual que la distancia mínima a otro vértice de *U*. En este caso, el vértice favorito es además un vértice candidato, pero no siempre ocurre esto, pudiendo haber vértices favoritos que no sean candidatos. El vértice C es también un vértice candidato (en este caso, la distancia a *R* es igual a la distancia mínima a otro vértice de *U*). El vértice B no es candidato, ya que existe una arista (la que conecta A y B) con menos peso que la arista que lo conecta a *R*.

El algoritmo de Prim adaptado a fronteras compuestas se implementa con un bucle. En cada iteración del mismo se realizan cinco pasos:

1. Relajar el peso mínimo de las aristas que unen cada vértice *i* de *U* con un vértice de *R*. Esta relajación consiste en calcular $e[i] = \min\{e[i], \omega(f,i)\}$ para cada vértice $i \in U$, $f \in F$.
2. Encontrar el *vértice favorito*.
3. Formar el nuevo conjunto *F* de vértices de la frontera compuesta, incluyendo inicialmente todos los *vértices candidatos*.
4. Añadir al conjunto *F* formado en el paso 3 el *vértice favorito* encontrado en el paso 2, obteniendo así la nueva frontera.
5. Actualizar el conjunto *U* de vértices no resueltos, eliminando del mismo todos los vértices que forman la frontera compuesta *F*.

El algoritmo de Prim adaptado a fronteras compuestas obtiene un árbol de recubrimiento mínimo. Si excluimos el tercer paso de los anteriores, el algoritmo se comporta igual que Prim, añadiendo únicamente un vértice a la frontera en cada iteración. Si añadimos además los vértices candidatos a la frontera, el algoritmo sigue siendo correcto, ya que no existe ninguna otra forma de añadir estos vértices al árbol con menor coste. Intuitivamente, hemos excluido la situación de la figura 3.4.2-1, que no se puede dar para vértices candidatos.

La mejora de este algoritmo es apreciable sobre todo al aplicarlo a grafos densos, en los que la probabilidad de que la frontera esté formada por un número más grande de vértices es mayor.

3.4.7. Pseudocódigo

Con la información proporcionada en el anterior apartado, no resulta difícil encontrar una implementación del algoritmo. Una posible es la siguiente:

```
void MST(c) {
  inicializa(e, padre, f, u);
  min = 0;
  mientras (min != INFINITO) {
    relaja(e, padre, f, u);
    (favorito, min) = minimo(e, u);
    candidatos(e, padre, f, u);
    f[favorito] = cierto;
    actualiza(f,u);
  }
}
```

La inicialización vuelve a contar con el vector `f` que indica qué vértices están en la frontera compuesta. El resto de la inicialización es idéntica al caso anterior.

```
void inicializa(e, padre, f, u) {
  para cada vértice i {
    e[i] = INFINITO;
    padre[i] = INFINITO;
    f[i] = falso;
    u[i] = cierto;

  }
  e[0] = 0;
  padre[0] = 0;
  f[0] = cierto;
  u[0] = falso;
}
```

La relajación ahora debe tener en cuenta que existen varias fronteras, y debe considerarlas todas a la hora de obtener la arista de menor peso.

```
void relaja(e, padre, f, u) {
  para cada i en paralelo {
    si(u[i]) {
      para cada j vecino de i {
        si(f[j]) {
          si(e[i] > w[j,i]) {
              e[i] = w[j,i];
              padre[i] = j;
          }
        }
      }
    }
  }
}
```

Formar la frontera inicial con todos los vértices candidatos se realiza también en paralelo para todos los vértices. Si en otras implementaciones eliminábamos los vértices de *U* tan pronto como los añadíamos a la frontera, en este caso no es posible ya que necesitamos recordar qué vértices estaban en *U* al terminar la iteración anterior del bucle.

```
void candidatos(e, padre, f, u) {
  para cada i en paralelo {
    f[i] = falso;
    si(u[i]) {
      candidato = e[i] < INFINITO;
      para cada j vecino de i {
        si(u[j]) {
          si(w[j,i] < e[i]) {
              candidato = falso;
          }
        }
      }
      si(candidato) {
        f[i] = cierto;
      }
    }
  }
}
```

Después de haber calculado todos los vértices que forman la nueva frontera, debemos actualizar, finalmente, el conjunto de vértices no resueltos *U*.

```
void actualiza(f, u) {
  para cada i en paralelo {
    si(f[i]) {
      u[i] = falso;
    }
  }
}
```

3.4.8. Implementaciones realizadas

Para la versión de Prim adaptada a fronteras compuestas, tenemos tantas formas de implementarlo como formas había de implementar el algoritmo de Dijkstra adaptado a fronteras compuestas, tratado en la sección anterior. Sin embargo, conociendo los

resultados que aquellas implementaciones dieron en términos de rendimiento, hemos podido filtrar las que sabemos que van a dar peor rendimiento.

En las implementaciones que utilizan una matriz de adyacencia como estructura de representación del grafo, se ha realizado solamente una implementaciones en GPU, que se corresponde con la implementación más directa del pseudocódigo mostrado en la sección anterior. Se han realizado, además, tres implementaciones en la CPU para contrastar los resultados (tanto su corrección como su rendimiento). La primera de ellas (CPU1) se corresponde con el algoritmo de Prim sin modificaciones. La implementación CPU2 es la implementación del algoritmo de Prim adaptado a fronteras compuestas de manera secuencial. Por último, CPU3 es una versión mejorada de CPU2, aunque las mejoras aplicadas en CPU3 (que se explican más adelante) no son aplicables a las implementaciones paralelas.

Para las implementaciones que usan una lista de adyacencia para representar el grafo, disponemos de dos implementaciones en CPU y dos en GPU. Las implementaciones CPU son las mismas que las implementaciones CPU1 y CPU2 del caso de las matrices de adyacencia. En el caso de las implementaciones paralelas, GPU1 y GPU2, se diferencian únicamente en el proceso de minimización. En GPU2, este proceso se realiza en dos etapas, mientras que en GPU1 se realiza en una única etapa.

Recordemos que estamos trabajando de nuevo con fronteras compuestas. Al igual que en el algoritmo de Dijkstra adaptado a fronteras compuestas, existen dos aproximaciones válidas para realizar el proceso de relajación: recorrer el conjunto de vértices no resueltos, o recorrer los vértices que componen la frontera. La implementación CPU3 es similar a la CPU2, salvo que en la relajación utiliza la segunda aproximación. Aplicar esta segunda aproximación en la GPU es, sin embargo, bastante complejo. El problema que en Dijkstra teníamos con posibles accesos concurrentes a la estimación de un vértice *i*, se ve agravado, ya que las variables que se escriben ahora son dos: el peso estimado y el vértice padre. La solución propuesta en el capítulo anterior para ese problema ya no nos sirve, y el SDK de CUDA no proporciona un método directo para crear una *sección crítica*.

En el manual del programador de CUDA **[3]** (en su sección 5.5) se dice explícitamente que para sincronizar hilos que pertenecen a distintos bloques (lo cual sería nuestro caso si queremos implementar una sección crítica), es necesario terminar la ejecución de un *kernel* y lanzar la ejecución de otro. Esto supondría para nosotros una solución muy pobre desde el punto de vista del rendimiento. Investigando, encontramos una forma de implementar esta sección crítica sin separar la ejecución en diferentes *kernels*. Sin embargo, esta implementación tiene dos problemas importantes: el primero es que no sigue las indicaciones del manual de programador de CUDA, al tratar de forzar la sincronización entre hilos de diferentes bloques sin finalizar la ejecución del *kernel*, y el segundo es que la implementación de dicha sección crítica se basa en la prioridad que CUDA adjudica a los hilos en caso de que sus caminos de ejecución diverjan. Estas prioridades no se definen explícitamente en **[3]** y son, por tanto, susceptibles de ser modificadas en futuras versiones.

Además, en cualquier caso, una sección crítica implica que el código se ejecuta secuencialmente, lo que siempre deteriora el rendimiento. Por estos motivos finalmente se desecharon las implementaciones basadas en este último tipo de relajación.

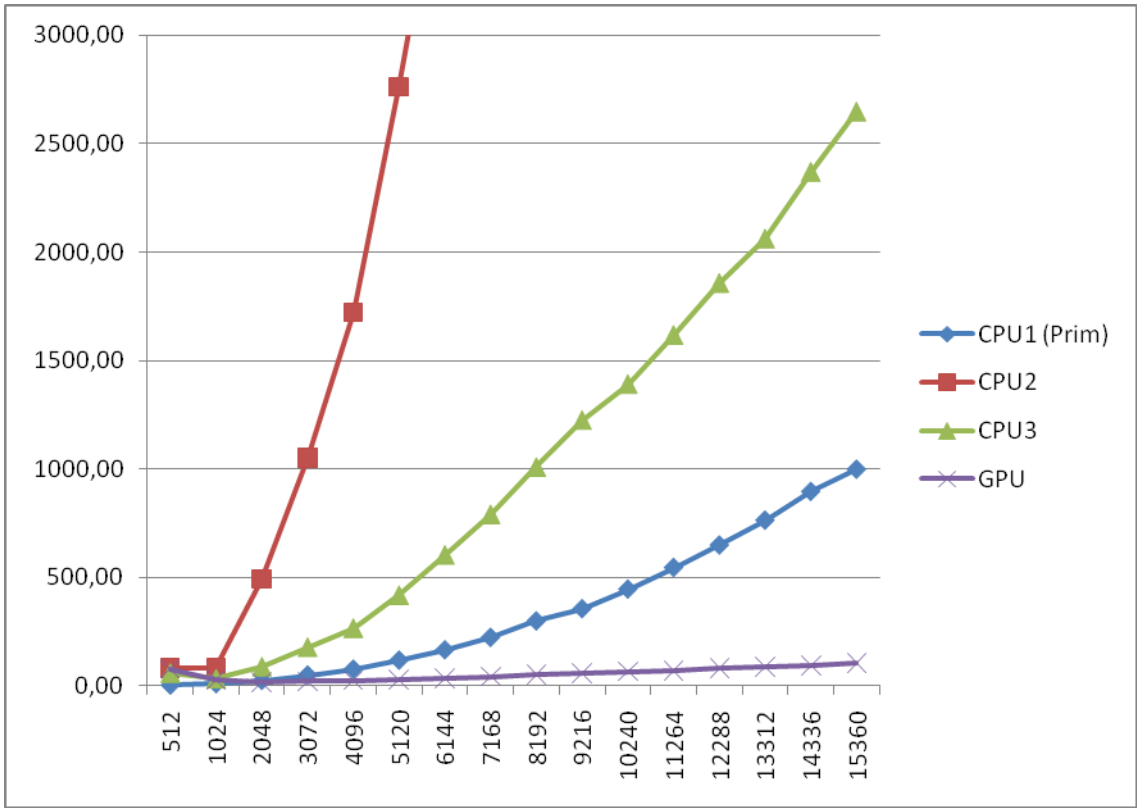
3.4.9. Medidas de rendimiento

La tabla siguiente muestra el tiempo (en milisegundos) invertido por cada una de las implementaciones que utilizan matrices de adyacencia para resolver el problema del MSP, para distintos tamaños de grafo. La última columna indica el *speedup* obtenido al utilizar la implementación GPU con respecto a la mejor de las implementaciones en CPU.

Nº vértices	CPU1 (Prim)	CPU2	CPU3	GPU	Ratio
512	2,63	82,90	58,82	78,76	0,03
1024	9,28	82,70	33,20	28,38	0,33
2048	24,09	492,25	88,33	16,94	1,42
3072	49,37	1051,83	178,53	20,93	2,36
4096	76,02	1722,13	266,11	25,38	2,99
5120	117,60	2760,37	417,50	30,34	3,88
6144	165,10	3934,74	603,30	35,71	4,62
7168	222,47	4981,43	789,98	42,85	5,19
8192	298,19	6286,05	1007,46	53,43	5,58
9216	354,59	7609,85	1224,45	57,41	6,18
10240	444,95	8720,23	1390,48	66,63	6,68
11264	544,76	10515,43	1616,32	71,66	7,60
12288	647,90	12642,07	1856,03	83,29	7,78
13312	762,31	15348,79	2059,83	87,41	8,72
14336	894,62	17827,55	2367,76	95,35	9,38
15360	996,95	20044,85	2645,28	103,97	9,59

Las cifras indican que, para un grafo de tamaño suficientemente grande (a partir de 2048 vértices), la ejecución en GPU es más rápida que en CPU. Esto es debido a que es más eficiente paralelizar la ejecución del algoritmo para grafos grandes que para grafos pequeños. Los resultados para las implementaciones CPU2 y CPU3 son malos, pero el objetivo de estas implementaciones no era obtener mejores tiempos, sino comprobar la corrección del algoritmo de Prim adaptado a fronteras compuestas antes de pasar a implementar dicho algoritmo en GPU.

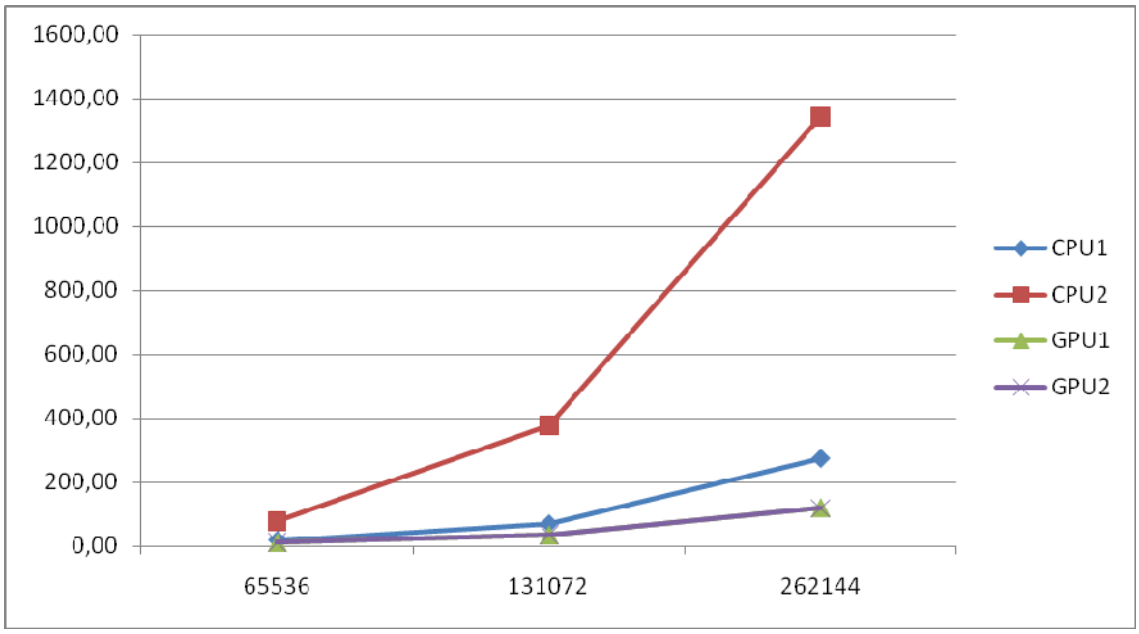
Los datos de la tabla anterior se pueden mostrar también en forma de gráfica, donde se observa mejor la evolución de los tiempos de ejecución con respecto al tamaño del grafo para cada una de las implementaciones. Como puede verse, la gráfica de CPU2 aparece truncada para que puedan apreciarse mejor las gráficas de CPU1 y GPU.



Para listas de adyacencia, las modificaciones aplicadas al algoritmo de Prim adaptado a fronteras compuestas dan un rendimiento inferior a la solución adoptada en la sección anterior. En cualquier caso, el rendimiento continúa siendo mejor que en las versiones secuenciales. La siguiente tabla indica los resultados obtenidos en las ejecuciones realizadas. La columna *Ratio* indica la ganancia obtenida con la implementación GPU1 con respecto a CPU1.

Nº vértices	CPU1 (Prim)	CPU2	CPU3	GPU1	GPU2	Ratio
65536	16,93	100,48	78,27	11,30	11,38	1,50
131072	68,06	420,86	377,35	34,80	34,41	1,96
262144	272,72	1762,04	1342,43	120,00	118,23	2,27

La siguiente gráfica muestra los datos anteriores de forma que se puedan apreciar de manera más visual.



De estos resultados puede deducirse que, en GPU, la aproximación basada en fronteras compuestas es mejor cuando se representan los grafos como matrices de adyacencia, mientras que la aproximación basada en frontera única permite obtener una mayor ganancia en la representación del grafo como listas de adyacencia.

3.5. APSP (All Pairs Shortest Path)

Dentro de los problemas clásicos de topología que se plantean sobre un grafo, presenta interés para nuestro trabajo el problema APSP (acrónimo de *All Pairs Shortest Path*).

Dado un grafo dirigido cuyas aristas están valoradas con números positivos, el objetivo del problema es calcular el coste del camino mínimo para cada par de vértices del mismo.

Como se puede observar es una extensión al problema SSSP tratado en la primera sección de este capítulo.

3.5.1. Versión clásica

Una de las soluciones a este problema la da el clásico *Algoritmo de Floyd*. Dicho algoritmo es recursivo y utiliza la técnica de programación dinámica en su diseño.

Vamos a explicar el razonamiento seguido para entender su funcionamiento y posteriormente veremos una implementación en pseudocódigo que nos permitirá deducir su coste asintótico.

Supondremos que el grafo $G=(V,E)$ nos viene dado por su matriz de adyacencia bidimensional *M*, donde cada una de sus dimensiones puede tomar valores de 0 a *N*-1, siendo $N=|V|$; de este modo tendremos que la matriz *M* es de la forma $M[0..N-1,0..N-1]$. Además suponemos que:

$$M[i,j]=\begin{cases} 0 & si\ i=j \\ coste & si\ hay\ arista\ de\ i\ a\ j \\ +\infty & si\ no\ hay\ arista\ de\ i\ a\ j \end{cases}$$

La función que utilizaremos será $C^k(i,j)$ cuyo significado será: coste mínimo para ir de i a j, pudiendo utilizar como vértices intermedios aquellos entre 0 y k.

La recurrencia, con $0 \le i,j,k \le N-1$, será elegir el mínimo coste entre el camino mínimo que pasa por el vértice k, y el que, hasta entonces, no pasa por él:

$$C^k(i,j)=mín\left\{C^{k-1}(i,j)\ ;\quad C^{k-1}(i,k)+C^{k-1}(k,j)\right\}$$

El caso básico lo tenemos cuando $k=-1$, es decir cuando no se utilizan vértices intermedios entre i y j.

$$C^{-1}(i,j)=\begin{cases} M[i,j] & si \quad i\neq j \\ 0 & si \quad i=j \end{cases}$$

Habremos conseguido nuestro objetivo cuando $k=N-1$, en ese momento la función nos devolverá el camino mínimo entre i y j, pues habremos considerado el paso por todos los vértices intermedios posibles.

El pseudocódigo para el algoritmo descrito sería el siguiente, suponiendo que la matriz de adyacencia que se recibe como dato se puede modificar y que en ella quedará almacenado el coste de cada camino mínimo entre pares de vértices:

```
void floyd( M[0..N-1, 0..N-1] ) {
  para k=0 hasta  N-1 {
    para i=0 hasta N-1 {
      para j=0 hasta N-1 {
        si (M[i,j]< M[i,k]+M[k,j]) {
          M[i,j] = M[i,k]+M[k,j];
        }
      }
    }
  }
}
```

Con cada finalización de una iteración exterior (del índice k) se completa un estado intermedio de la solución. Dicha solución parcial contendrá los caminos mínimos para ir de i a j, habiendo considerado el paso por los primeros k vértices.

Como se puede observar tenemos tres bucles anidados de N iteraciones cada uno, con lo cual es fácil concluir que el coste de este algoritmo es $\Theta(N^3)$.

3.5.2. Versiones paralelas

La forma de paralelizar el algoritmo clásico secuencial de Floyd es relativamente conocida. Consiste en dividir la matriz M en sub-matrices con el fin de ejecutar los dos bucles interiores en paralelo.

Por lo dicho anteriormente el bucle exterior no se puede paralelizar, porque tiene que estar completada la iteración anterior para poder seguir.

Para ver la eficiencia de las distintas formas de descomponer la matriz total del grafo, hemos decidido hacer distintas implementaciones que permitan comparar sus tiempos de ejecución.

Para este problema no habrá distintas versiones para listas de adyacencia y matriz de adyacencia, ya que el algoritmo elegido y la representación de la salida nos obligan a representar el grafo mediante una matriz. Por tanto, si se deseara ejecutar el algoritmo con un grafo representado con listas de adyacencia, tenemos dos opciones.

La primera consiste en transformar la representación del grafo a matriz y ejecutar alguno de los algoritmos que vamos a ver a continuación. La segunda opción propone una alternativa diferente, resolver el problema APSP ejecutando N veces (una para cada vértice del grafo) un algoritmo capaz de resolver el problema SSSP; a esta técnica la denominamos N-SSSP.

La segunda opción nos llevaría a una interesante comparativa de rendimiento entre dos algoritmos diferentes, que podría servir también para paliar las limitaciones explicadas en la sección 3.5.3. Suponemos que la topología del grafo tendrá una notable incidencia en el rendimiento del algoritmo N-SSSP, aunque contamos con numerosas versiones que resuelven SSSP. Llevar a cabo un análisis exhaustivo para comparar el rendimiento de APSP vs N-SSSP, requeriría un tiempo excesivo para las expectativas de este trabajo y hemos decidido dejarlo como trabajo futuro.

3.5.2.1. División por bloques

La primera implementación paralela realizada consiste en partir la matriz M en submatrices cuadradas más pequeñas. Cada submatriz tendrá un tamaño determinado por una constante **LADO_CUAD**, que en nuestro caso vale 16.

La granularidad de cada hilo será a nivel de celda. Es decir que cada hilo tendrá asignada la tarea de calcular un mínimo para la componente $M[i, j]$ correspondiente, en cada una de las iteraciones.

Cada bloque de hilos estará compuesto por exactamente $16 \times 16 = 256$ hilos. Por tanto cada submatriz de M tendrá dimensiones 16×16. Luego la rejilla de bloques tendrá dos dimensiones, de **nVertices/LADO_CUAD** cada una. Por lo tanto, el número de vértices del grafo tendrá que ser múltiplo de **LADO_CUAD**.

En la implementación esto está especificado como sigue:

```
dim3 dimBlock( LADO_CUAD, LADO_CUAD);
dim3 dimGrid( nVertices/LADO_CUAD, nVertices/LADO_CUAD);
```

Luego el *kernel* que ejecuta la actualización para la matriz M, se ejecuta N veces. Siempre con la misma configuración.

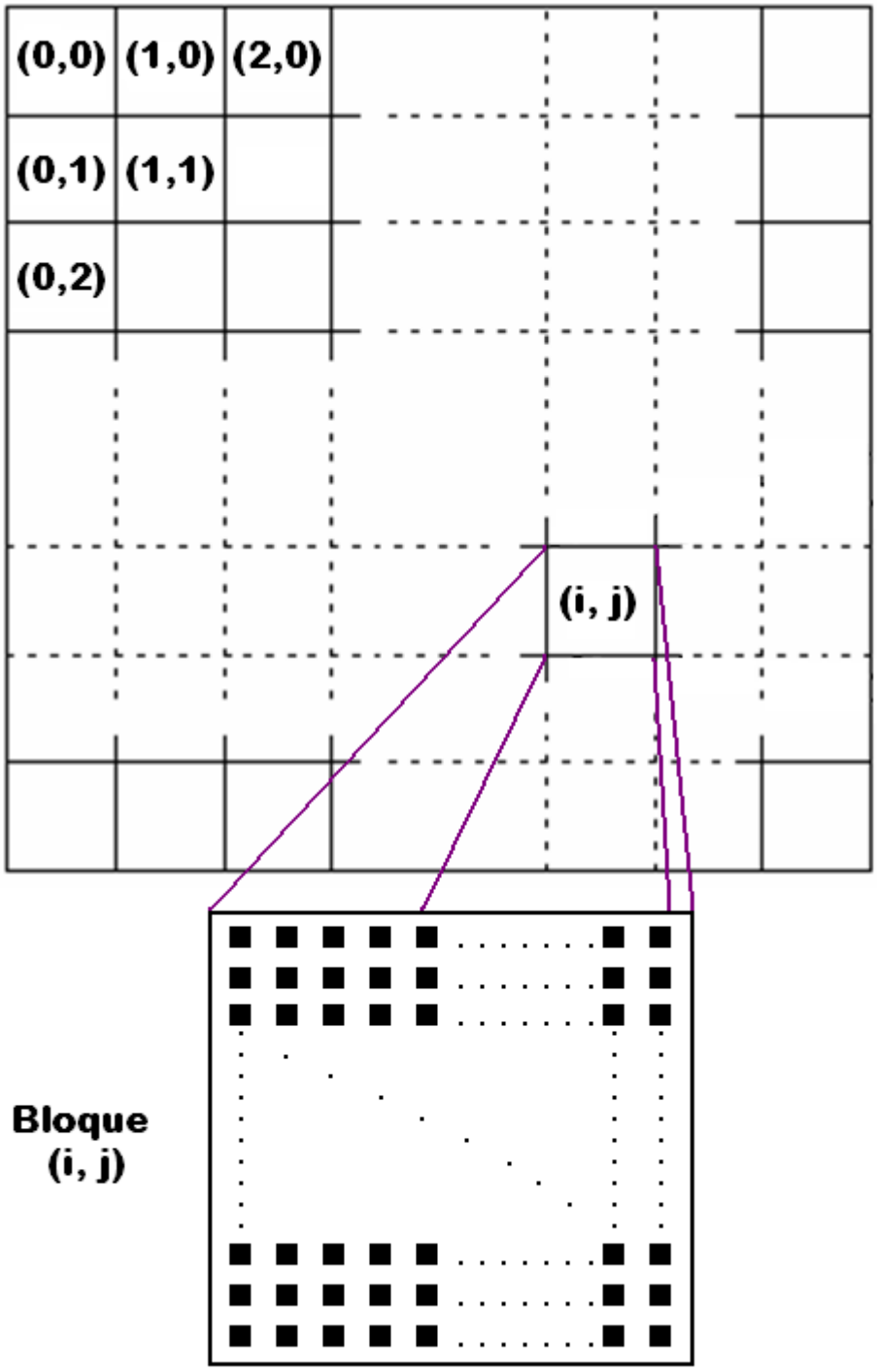


Figura 3.5-1. Partición de la matriz M en submatrices cuadradas

Para hacer el acceso a memoria más eficiente, los hilos realizan un almacenamiento de la fila k y la columna k en memoria compartida. Observemos que, para cada bloque, se necesitan 16 valores de la fila k y otros 16 valores de la columna k, lo que totalizan 32 datos.

Si accediéramos a memoria global con cada hilo se trataría de 256 accesos. Como se comentó en la sección 2.2.2, la memoria compartida ofrece un acceso con menor latencia y la posibilidad de leer varios bancos de memoria a la vez. Por ello, optamos por hacer que

los hilos con componente $x = 0$ leyeran la columna k, y los hilos con componente $y = 0$, la fila k y la almacenaran en memoria compartida para que su lectura posterior fuera más rápida.

3.5.2.2. División por filas

En esta implementación intentamos cambiar la división que hacíamos en el caso anterior.

Nuevamente repetimos la granularidad de los hilos como en el algoritmo anterior, cada hilo se encarga de calcular el mínimo de la celda $M[i, j]$ de la matriz. Mantenemos también el tamaño del bloque que va a seguir siendo 256 (definido por la constante **`TAM_BLOQUE`**), aunque esta vez será unidimensional y se extenderá a lo largo de las filas.

La nueva estrategia de división de la matriz consiste en que cada fila de la matriz estará dividida en **`nVertices/TAM_BLOQUE`** bloques. En este caso el número de vértices del grafo deberá ser múltiplo de **`TAM_BLOQUE`**.

Dicha división está especificada como sigue:

```
dim3 dimBlock( TAM_BLOQUE);
dim3 dimGrid( nVertices/TAM_BLOQUE, nVertices);
```

El *kernel* se ejecuta N veces con la misma configuración. Aunque haremos un análisis más detallado en la sección de rendimiento, obsérvese que el número de bloques y de hilos es igual que en la implementación anterior.

Con esta organización se consigue que el acceso a la columna k, se haga a través de una única variable en memoria compartida, que a los efectos de lectura será como leer de un registro, según vimos en el capítulo 2.

Por el contrario, se pierde el acceso por memoria compartida a la fila k, que ahora hay que leer siempre de memoria global. Existe una opción para almacenar la fila k completa como una textura en el espacio de texturas y leer desde ahí que, al estar cacheado, nos dará un mejor rendimiento. El problema es que en cada una de las N iteraciones hay que copiar el valor de la fila a dicho espacio de memoria.

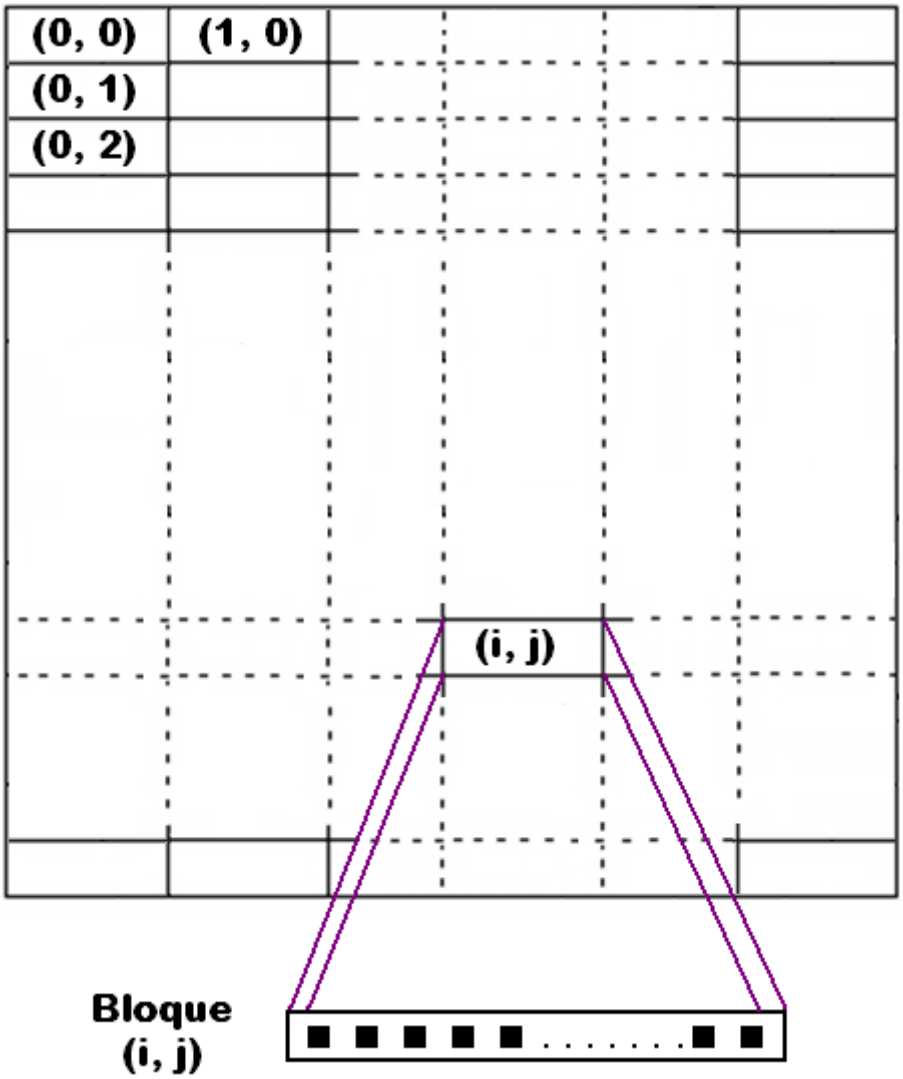


Figura 3.5-2. Partición de la matriz M en filas.

3.5.2.3. Bucle por filas

La tercera alternativa consiste en disminuir drásticamente el número de hilos en ejecución.

Para conseguirlo, debemos asignar más tarea a cada hilo, es decir, que un mismo hilo se encargue de calcular más mínimos (hasta ahora sólo calculaba uno). Ahora un hilo se encargará de actualizar toda una fila completa de la matriz.

El tamaño de los bloques seguirá determinado por **`TAM_BLOQUE`**, de forma que un bloque se ocupará de actualizar 256 filas consecutivas de la matriz. Como siempre, el *kernel* se ejecuta N veces con la misma configuración.

La división de la matriz está determinada de la siguiente forma:

```
dim3 dimBlock(1, TAM_BLOQUE);
dim3 dimGrid(1, nVertices/TAM_BLOQUE);
```

Otra vez la columna k se verá reducida a una variable, pero será distinta para cada hilo del bloque.

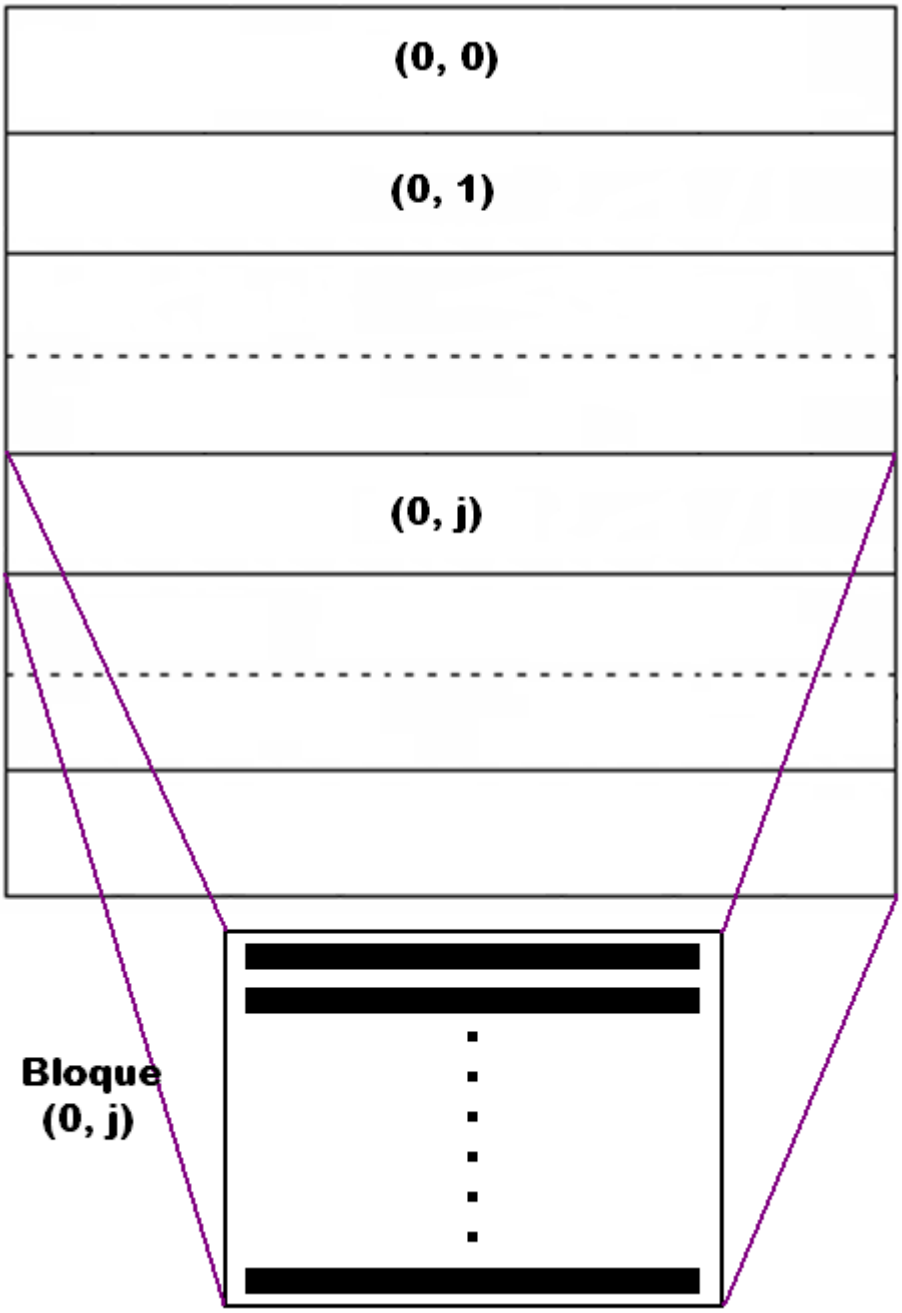


Figura 3.5-3. Partición de la matriz M en filas de hilos.

En esta implementación también obtenemos una optimización usando el espacio de memoria compartida. Cada **TAM_BLOQUE** iteraciones del bucle que recorre la fila asignada al hilo, se guardan en un array compartido, **TAM_BLOQUE** posiciones de la fila k (como es lógico, cada hilo guarda una posición), para que después sean leídas de una vez por todos los hilos.

3.5.2.4. Bucle por columnas

La idea es exactamente la misma que en la implementación anterior, pero esta vez, en lugar de que un hilo se encargue de actualizar una fila completa de la matriz, actualizará una

columna. Todo lo explicado para el caso anterior es totalmente simétrico para esta implementación.

Con esto se obtendrá un mejor rendimiento ya que las celdas $M[i, j]$ e $M[i+1, j]$ se encuentran en posiciones de memoria contigua.

Los bloques seguirán estando compuestos por 256 hilos y el *kernel* se ejecutará N veces con la misma configuración.

La división de la matriz estará determinada de la siguiente forma:

```
dim3 dimBlock( TAM_BLOQUE, 1);
dim3 dimGrid( nVertices/TAM_BLOQUE, 1);
```

Ahora será la fila k la que se vea reducida a una variable, pero será distinta para cada hilo del bloque.

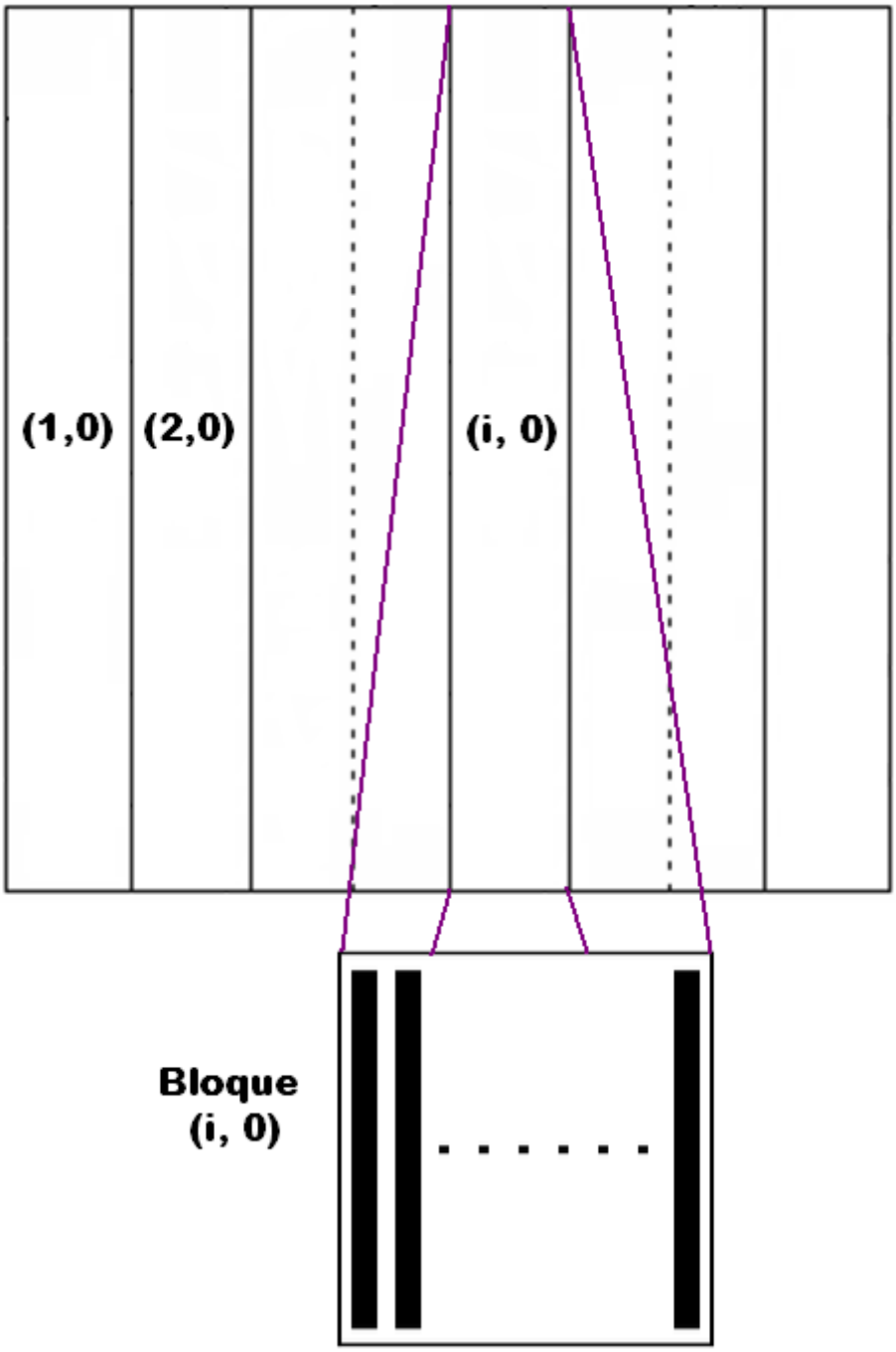


Figura 3.5-4. Partición de la matriz M en columnas de hilos.

En esta implementación también tenemos una optimización si usamos el espacio de memoria compartida. Cada **TAM_BLOQUE** iteraciones del bucle que recorre la columna asignada al hilo, se guardan, en un array compartido, **TAM_BLOQUE** posiciones de la fila *k* (de nuevo, como es lógico, cada hilo guarda una posición), para que después sean leídas de una vez por todos los hilos.

3.5.3. Limitaciones

Ejecutar cualquiera de los algoritmos propuestos en la sección anterior conlleva una limitación. La totalidad de la matriz de adyacencia que representa el grafo tiene que estar almacenada en memoria.

Considerando que el espacio requerido para la matriz es $\Theta(N^2)$ tenemos un crecimiento considerablemente rápido de la demanda de memoria.

Realizando las pruebas de ejecución en una tarjeta gráfica GTX280 de 1GB de memoria, conseguimos ejecutar con éxito pruebas para grafos de hasta **11.264** vértices.

Queda como tarea futura la implementación de un algoritmo más elaborado que no presente esta limitación y vaya almacenando en la memoria de la tarjeta gráfica, submatrices que quepan. Los resultados producidos entonces, se actualizarían en el *host*.

Asimismo, como veremos en la siguiente sección, el tamaño de los grafos admitidos por nuestra tarjeta gráfica de pruebas nos permite analizar con holgura la mejora de rendimiento que supone utilizar cualquiera de los algoritmos paralelos que hemos implementado.

3.5.4. Medidas de rendimiento

3.5.4.1. Hilos y bloques de cada versión

Tomamos las constantes citadas en las explicaciones: **`LADO_CUAD = 16`**, **`TAM_BLOQUE = 256`**. El número de vértices varía con el grafo de modo que ciertos resultados quedarán expresados en función de ese parámetro.

Versión	Dimensión bloque	Hilos por bloque	Dimensión rejilla	Bloques totales	Hilos totales
Bloques cuadrados	16 x 16	256	$\frac{nv}{16}\cdot\frac{nv}{16}$	$\left(\frac{nv}{16}\right)^2$	nv^2
Bloques por filas	256 x 1	256	$\frac{nv}{256}\cdot nv$	$\left(\frac{nv}{16}\right)^2$	nv^2
Bucles por filas	1 x 256	256	$\frac{nv}{256}$	$\frac{nv}{256}$	nv
Bucles por columnas	256 x 1	256	$\frac{nv}{256}$	$\frac{nv}{256}$	nv

Como comentamos, las últimas dos implementaciones tienen menos hilos, debido a que se aumenta la tarea que realiza cada uno. En las primeras dos versiones un hilo actualiza el mínimo de una sola celda de la matriz; en las dos últimas, un hilo actualiza toda una fila o columna, respectivamente.

3.5.4.2. Ejecuciones

Para hacer las pruebas de ejecución y la medida de tiempos, se generaron grafos aleatorios con el número de vértices indicado. En todos ellos, todos los vértices tienen grado 5; en todo caso, este dato es irrelevante ya que el número de arcos no influye en el coste del algoritmo.

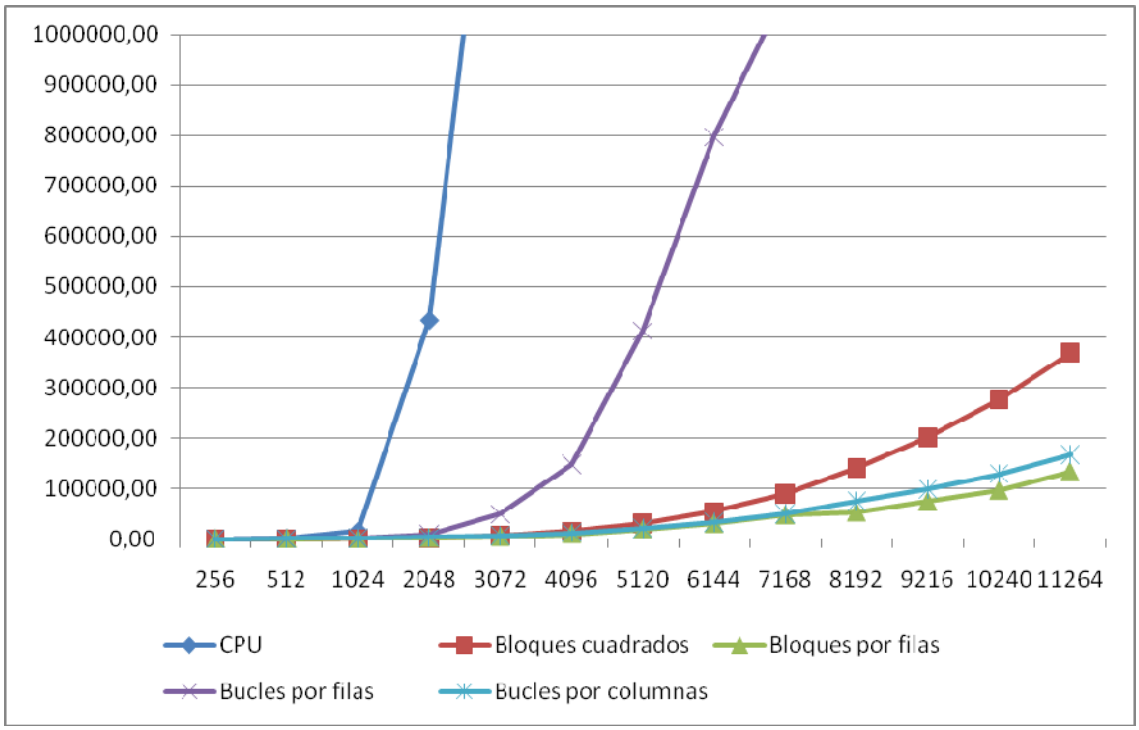
Para cada grafo de distinto tamaño, se realizaron 3 ejecuciones. Lo que aparece en las casillas es la media aritmética de tiempos de las mismas.

Nº vértices	CPU	Bloques cuadrados	Bloques por filas	Bucles por filas	Bucles por columnas
256	98,33	12,22	5,19	84,20	40,71
512	889,03	38,23	22,96	377,72	160,80
1024	15703,27	285,84	198,63	1671,46	691,02
2048	434073,84	1962,01	1309,01	8239,16	2814,85
3072	1617488,34	6365,85	4073,12	49223,78	6573,67
4096	3954671,31	14952,32	9252,91	147133,73	12280,32
5120	(*)	30179,05	17672,95	411948,56	20525,43
6144	(*)	53943,67	29898,32	796989,19	32957,90
7168	(*)	89530,16	47058,19	1087700,94	49588,17
8192	(*)	140478,93	52510,55	(**)	74284,02
9216	(*)	201033,71	73655,57	(**)	98746,29
10240	(*)	276628,44	95810,34	(**)	128794,30
11264	(*)	368827,80	132584,03	(**)	166907,19

(*) La última ejecución del algoritmo secuencial tarda 66 minutos, frente a la peor versión en GPU que tarda 2,5 minutos aproximadamente, lo que es una mejora de tiempo más que considerable, que seguiría ascendiendo al aumentar el número de vértices. Decidimos pues, no continuar con la ejecución del algoritmo secuencial.

(**) La versión GPU de bucles por filas, es claramente peor que las otras tres. Su tiempo de ejecución, en la prueba de 7168 vértices, es dos órdenes de magnitud superior al de las otras versiones y perdía interés continuar con la pruebas.

Además, si se prolonga la duración de un hilo por mucho tiempo, el sistema operativo interviene y mata los hilos, con lo cual las ejecuciones no concluyen.



En la gráfica anterior se puede ver la evolución del tiempo de ejecución según aumenta el tamaño del grafo. Las líneas correspondientes a las ejecuciones en CPU y la ejecución en GPU utilizando la versión de bucles por filas aparecen truncadas para que se puedan apreciar mejor las diferencias entre el resto de implementaciones.

4. Otras aplicaciones de CUDA en programación de propósito general

Hasta aquí hemos visto la aplicación de CUDA en algoritmos de un área específica, los grafos. Dado que existen otras áreas que utilizan algoritmos con características que permiten su paralelización, queremos también dedicar un análisis a algunas de las que nos resultaron de interés.

Hay tres factores principales por los cuales hemos elegido los diferentes temas que se desarrollarán en este capítulo. El primero, por tener conocimientos previos en el área del problema (aunque fueran básicos); el segundo, por la relevancia del área; y el tercero, por la naturaleza de los algoritmos, ya que buscamos que admitieran una forma paralela sencilla.

También hemos establecido una colaboración con otro grupo de Sistemas Informáticos, que nos brindaron información de su proyecto sobre *Visión Estereoscópica* para poder crear versiones en CUDA de los algoritmos que ellos implementaron en Java.

4.1. Programación Evolutiva y Algoritmos Genéticos

Los algoritmos evolutivos (EAs) son métodos de búsqueda estocástica en numerosos problemas de búsqueda y optimización. A diferencia de muchas otras técnicas de optimización, los EAs mantienen una población de posibles soluciones que se manipulan mediante ciertos operadores para llegar a una solución satisfactoria.

Los algoritmos genéticos son uno de los tipos de algoritmos evolutivos más extendidos. Por este motivo, nos centraremos en ellos y analizaremos la manera de adaptarlos al modelo de programación de CUDA.

4.1.1. Algoritmo genético

La biología evolutiva ha inspirado las técnicas utilizadas por los algoritmos genéticos. En ellos se aplican conceptos de esa disciplina como la herencia, la mutación, la selección y la recombinación.

Inicialmente, se genera de manera aleatoria un conjunto de posibles soluciones que se denomina *población inicial*. Estas posibles soluciones son los *individuos* de la población. El algoritmo genético evalúa cada individuo de la población inicial utilizando cierta función de adaptación o *fitness*. A continuación, se repite un proceso de constitución de generaciones sucesivas. La población de cada nueva generación se construye sobre la población de la generación anterior. En el proceso de *selección* se eligen los individuos de la población anterior que permanecerán en la población de la nueva generación. Los individuos con mejor adaptación tendrán más posibilidades de permanecer que aquellos en los que sea peor. A los individuos seleccionados se les aplica una serie de modificaciones. En los algoritmos genéticos estas modificaciones suelen ser el cruce y la mutación. El cruce consiste en seleccionar parejas de individuos que, con cierta probabilidad, van a combinarse para generar otras posibles soluciones que llamaremos *descendencia*. La mutación es un operador que se aplica a cada individuo y, con cierta probabilidad, altera parcialmente dicha

solución obteniéndose una solución diferente. Por último, se evalúa la población de la nueva generación.

Este proceso se repite hasta que se ha alcanzado cierto número de generaciones o bien hasta que se cumplen determinadas condiciones de parada. En este estudio no vamos a profundizar en las condiciones de parada aplicables al algoritmo genético, por lo que repetiremos el proceso un número de generaciones preestablecido.

Un algoritmo genético suele tener pues un aspecto similar al siguiente:

```
t = 0;
inicializar(P(t));
evaluar(P(t));
mientras( t < MAX_GENERACIONES ) {
   t = t + 1;
   P(t) = selección(P(t-1));
   cruce(P(t));
   mutacion(P(t));
   evaluar(P(t));
}
```

Llamamos `P(t)` a la población de la generación `t`. Dicha población se genera mediante un proceso de selección a partir de la población de la generación anterior, `P(t-1)`. Sobre esa población se aplican los operadores de cruce y mutación, explicados más arriba.

4.1.2. Versiones paralelas

La búsqueda del paralelismo en los algoritmos evolutivos y, en particular, en los algoritmos genéticos, es un tema sobre el que ya se ha realizado mucha investigación. En **[6]** se realiza un estudio exhaustivo de los diferentes tipos de algoritmos evolutivos paralelos, y en **[7]** se presentan mejoras en la paralelización de algoritmos genéticos.

La ejecución de un algoritmo genético, cuando el número de individuos de la población es muy numeroso o el tamaño de cada individuo es muy grande, requiere gran capacidad de cómputo. Adoptar un modelo de programación en paralelo como CUDA permite mejorar el rendimiento distribuyendo la carga de trabajo entre varias unidades de cómputo. Además, en **[6]** se concluye que el uso de algoritmos evolutivos paralelos (PEAs) conduce a mejores resultados incluso en su ejecución sobre un único procesador. Esto es debido a la distribución que se hace de los individuos en una *población estructurada*.

En los tipos de algoritmos genéticos que hemos explorado, todos los operadores se aplican globalmente, es decir, cualquier individuo puede, potencialmente, combinarse con cualquier otro para generar descendencia. Igualmente, la selección se aplica a todos los individuos de la población. Existe otro modelo en el que los individuos poseen cierto orden espacial, dando lugar a algoritmos genéticos estructurados.

Entre los tipos de algoritmos genéticos (en adelante GA, de *Genetic Algorithms*) estructurados más extensamente conocidos están los GA distribuidos (dGA) y los GA celulares (cGA). El modelo del dGA consiste en un conjunto de subpoblaciones (islas), en las cuales los individuos evolucionan por separado y, ocasionalmente, intercambian individuos. Este modelo es el más popular debido a que es el más adecuado para procesadores *multi-core*. Sin embargo, en nuestro caso, el modelo del cGA es el que mejor se adapta a CUDA.

La principal diferencia de los cGA respecto del tipo de algoritmo genético explicado inicialmente es que los procesos de selección, cruce y mutación están completamente descentralizados. Los individuos de la población se sitúan en las diferentes posiciones de una rejilla bidimensional (aunque éste es el caso más común, la rejilla podría ser de una, dos o tres dimensiones). Los procesos mencionados se realizan de manera independiente para cada uno de los numerosos vecindarios. Cada individuo posee su propio vecindario de individuos, formado por él mismo y por los individuos adyacentes en la rejilla, con los que puede interactuar; a la vez, un mismo individuo forma parte de varios vecindarios.

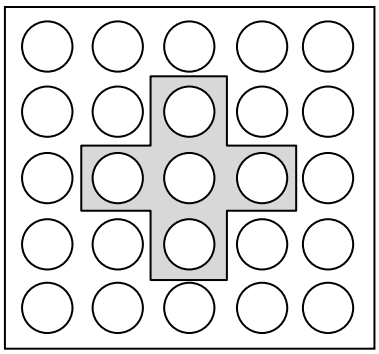


Figura 4.1-1. Rejilla de un cGA y, sombreado en gris, un vecindario

En los cGA las poblaciones tienen un único individuo y además, dicho individuo está fuertemente ligado a los demás individuos de su vecindario.

En **[7]** se presenta la implementación de un algoritmo genético celular simple. En nuestro estudio partimos de dicho algoritmo, realizando las modificaciones adecuadas para adaptarlo al modelo de programación de CUDA. El resultado de esas transformaciones es el siguiente algoritmo cGA:

```
t = 0;
para cada individuo de P(t) en paralelo {
   inicializar(individuo);
   evaluar(individuo);
}
mientras( t < MAX_GENERACIONES ) {
   t = t + 1;
   para cada individuo de P(t) en paralelo {
      padre1 = seleccion_local(individuo, P(t-1));
      padre2 = seleccion_local(individuo, P(t-1));
      aux = cruce(padre1, padre2);
      aux = mutacion(aux);
      evaluar(aux);
      si( mejor(aux, individuo) ) {
         individuo = aux;
      }
   }
}
```

La creación de la población inicial se realiza en paralelo, aunque el proceso de generación de cada individuo no varía. Es en el cuerpo del bucle principal donde aparecen las principales diferencias con respecto a la versión presentada en un principio. Para cada uno de los individuos de la población se seleccionan dos individuos de su vecindario, teniendo más posibilidades de ser seleccionados aquéllos que tengan mayor valor de adaptación. Los dos individuos seleccionados se combinan para generar otra posible

solución que almacenamos en `aux`. Después se aplica el operador de mutación a `aux` que, con probabilidad de mutación P_m, la modifica parcialmente. Tras evaluar la nueva solución generada, si ésta es mejor que el individuo inicial, este individuo se sustituye por la nueva solución.

4.1.3. Aplicación a un problema concreto: el problema del viajante

El algoritmo cGA que hemos implementado en CUDA resuelve el problema del viajante (*Travelling Salesman Problem*, TSP). Este problema consiste en, dadas N ciudades de un territorio, encontrar una ruta que, empezando y terminando en cierta ciudad concreta, pase una sola vez por cada una de las ciudades y minimice la distancia recorrida por el viajante.

En este caso, las posibles soluciones son permutaciones $P = \{c_0, c_1, \ldots, c_{N-1}\}$ del conjunto de ciudades. La búsqueda se puede plantear como una minimización, donde el valor a minimizar es

$$d_P = \sum_{i=0}^{N-1} d\left[c_i, c_{(i+1) \bmod (N)}\right]$$

donde $d[i,j]$ representa la distancia entre las ciudades i y j. Las ciudades van a estar representadas por números enteros entre 0 y $N-1$, y vamos a asumir que, en todo caso, $c_0 = 0$.

Los operadores de selección, cruce y mutación utilizados son bastante comunes en los algoritmos genéticos que trabajan con permutaciones. En este texto no se profundizará en el funcionamiento de estos operadores, aunque se puede encontrar una descripción detallada de todos estos métodos en **[8]**.

Para la implementación de un algoritmo genético es necesario disponer de un generador de números pseudo-aleatorios. En el SDK de CUDA se incluye un ejemplo de generación de números pseudo-aleatorios en GPU, que hemos adaptado para utilizarlo para este GA.

El operador de selección utiliza el método de selección por ruleta para obtener, de los cinco individuos que componen cada vecindario, los dos progenitores de la nueva generación. Dichos progenitores se combinan utilizando el método de cruce por emparejamiento parcial (*Partially Mapped Crossover*, PMX) para formar la descendencia. Por último, se utiliza el método de mutación por inversión, que se aplica a cada individuo con una probabilidad de 0,2.

El modelo de programación de CUDA nos permite trabajar con un gran número de individuos sin que el rendimiento se resienta. En la versión implementada, la población estaba formada por un total de 16384 individuos.

En cuanto al tamaño del problema, es decir, el número N de ciudades a recorrer, se trata de un parámetro modificable y que variaremos a lo largo de las pruebas con el fin de comparar la eficacia y rendimiento del algoritmo. El número de iteraciones aumenta proporcionalmente con N para tratar de mantener la eficacia del algoritmo cuando el tamaño del problema (y, por tanto, del espacio de búsqueda) sea muy grande.

4.1.4. Medidas de rendimiento

En general, la ganancia en velocidad de un algoritmo paralelo es una manera extendida de medir su eficiencia. Sin embargo, aunque es un sistema muy extendido en el campo de los algoritmos paralelos deterministas, es difícil adaptar este sistema a los algoritmos genéticos que son, por naturaleza, indeterministas.

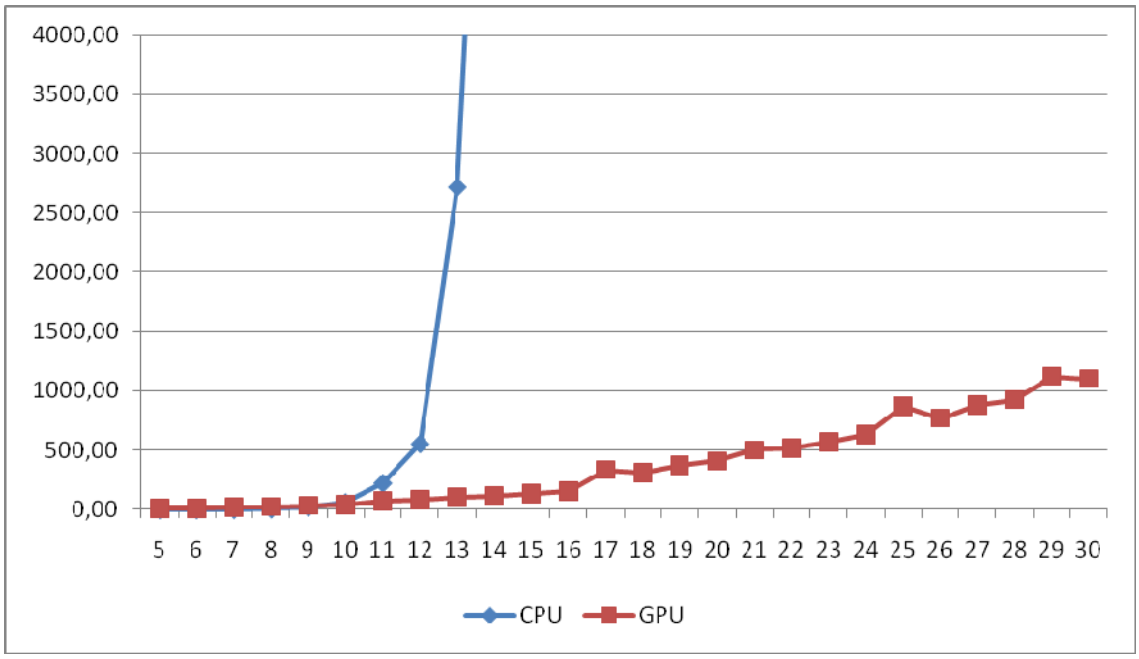
Para poder comparar las soluciones aportadas por el algoritmo genético con las soluciones óptimas, se ha implementado también un programa determinista que calcula dichas soluciones utilizando la técnica de ramificación y poda. Aunque se incluyen los tiempos empleados por dicho programa para sus cálculos, no tiene sentido compararlos con los tiempos empleados por el algoritmo genético con el fin de medir su eficiencia.

Los tiempos empleados por el algoritmo genético para la resolución del problema varían bastante, por lo que se han realizado numerosas ejecuciones para cada uno de los tamaños del problema probados. En este caso, se ha resuelto el problema del viajante para un número *N* de ciudades entre 5 y 30. Para cada valor de *N* se han realizado 25 ejecuciones. La siguiente tabla muestra los tiempos medios invertidos por la implementación determinista y la implementación utilizando un algoritmo genético en GPU. También se incluye una columna con los cocientes medios entre los costes de las soluciones utilizando GAs y el coste mínimo. Para valores de *N* pequeños, estos cocientes son siempre 1, lo cual quiere decir que la solución proporcionada por el GA es óptima.

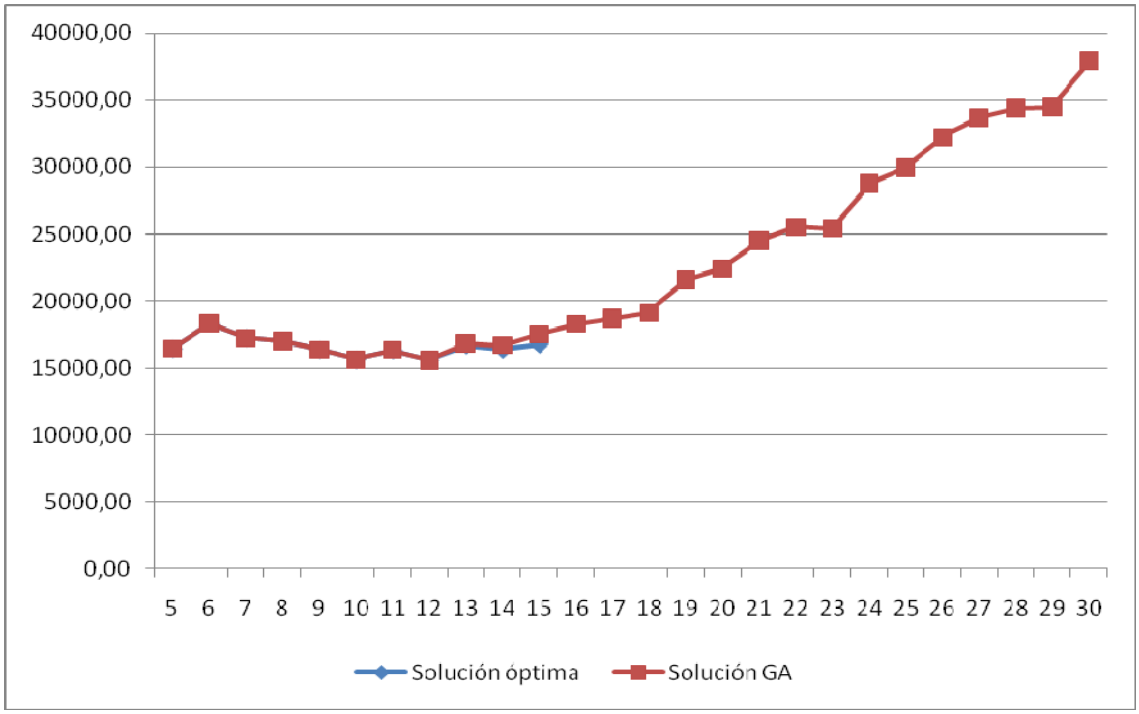
Nº vértices	CPU	GPU	C. Resultados
5	0,11	5,99	1,0000
6	0,45	8,76	1,0000
7	1,50	17,08	1,0000
8	6,08	22,74	1,0000
9	21,26	33,07	1,0000
10	62,13	37,12	1,0000
11	224,45	67,00	1,0000
12	555,81	79,24	1,0000
13	2716,50	100,79	1,0107
14	8714,58	111,78	1,0202
15	31309,84	134,69	1,0443
16	(*)	151,88	(*)
17	(*)	333,62	(*)
18	(*)	306,16	(*)
19	(*)	372,03	(*)
20	(*)	408,41	(*)
21	(*)	501,75	(*)
22	(*)	513,44	(*)
23	(*)	566,47	(*)
24	(*)	632,48	(*)
25	(*)	867,89	(*)
26	(*)	769,68	(*)
27	(*)	876,88	(*)
28	(*)	923,69	(*)
29	(*)	1122,04	(*)
30	(*)	1101,82	(*)

(*) La ejecución del programa determinista que resuelve el problema es muy costosa tanto en tiempo como en memoria, y sólo se han podido obtener soluciones de referencia para valores de *N* de hasta 15 ciudades.

En la gráfica siguiente se muestra la evolución del rendimiento de ambas ejecuciones según aumentamos el tamaño del problema. Para que se pueda apreciar mejor la evolución del algoritmo genético, los tiempos de la solución determinista aparecen truncados.



Por último, mostramos en una nueva gráfica el coste de los resultados proporcionados por el GA en comparación con las soluciones óptimas. Se puede ver que la solución del GA está siempre muy próxima a la referencia.



4.2. Cálculo del PageRank

4.2.1. Vigencia del problema

Día a día el número de documentos en Internet aumenta de forma incalculable y los buscadores se enfrentan a un gran desafío: mejorar la calidad de sus búsquedas.

Para un buscador tipo, hallar resultados para una búsqueda determinada no supone el mayor de los escollos. Uno de los problemas más difíciles que un buscador debe resolver es el **posicionamiento de los resultados** obtenidos en respuesta a una pregunta, de acuerdo a los criterios de los usuarios.

Nadie puede dudar de que la importancia de los buscadores de Internet es vital, ya que son una herramienta de uso cotidiano para cualquier usuario de la red. Por tanto el modelo que siga para posicionar los resultados puede determinar el éxito del buscador.

4.2.1.1. Modelo de la red

Para conseguir el tratamiento de la información que hay en la Red es necesario contar con una forma de modelarla. La más difundida es imaginar la red como un grafo dirigido, objeto principal de estudio de este proyecto.

El grafo de la red es un grafo dirigido con las siguientes características:

- sus vértices son páginas web,
- sus aristas son enlaces,
- si existen múltiples enlaces de una página a otra se considerará un único enlace,
- no se consideran enlaces a la propia página.

Dependiendo del estudio que se quiera hacer se pueden considerar grafos con distinto nivel de detalle: páginas, sitios o dominios.

El grafo sirve para analizar distintas características de la red y obtener estadísticas como pueden ser: número de enlaces entrantes y salientes de una página, cálculo de una componente fuertemente conexa del grafo, número de nodos sin enlaces, diámetro medio de la red (distancia media en número de enlaces entre dos páginas cualesquiera) y cálculo del *PageRank*.

Este cálculo se basa en características del grafo como las siguientes:

- Es común que dos documentos tengan enlaces recíprocos.
- Las páginas y los enlaces se pueden crear y borrar (hay dinamismo).
- El grafo tiene componentes (subgrafos) fuertemente conexas. Son grupos de URLs muy relacionados entre sí y poco relacionados con el resto de la red.
- La mayor parte de los enlaces (80%) son al propio sitio.
- El grafo de la red sigue una *"power law distribution"*. Se cumple la propiedad de que el número de vértices con grado k es proporcional a $K^{-\beta}$. Con ello se quiere decir que, en este tipo de grafos, existen unos pocos vértices con grado muy alto y muchos vértices con grado pequeño.

La *medida de la relevancia de una página* basada en la estructura del grafo de la red se fundamenta en la suposición de que si una página A tiene un enlace a una página B, es que la página A está recomendando la página B. Por lo tanto, si existen muchos enlaces a una página, la página está muy recomendada y por lo tanto tendrá más calidad.

La calidad de una página será proporcional pues a la calidad de la página que la apunta e inversamente proporcional al número de enlaces salientes de la página que la apunta.

4.2.2. Versión clásica

A nivel teórico, el modelo considerado para el cálculo del PageRank se denomina *"random surf model"* y se basa en la construcción de una cadena de Markov. Este modelo simula el comportamiento de un *"surfer"* que se mueve a través de la red siguiendo los enlaces de las páginas. La probabilidad de que se traslade de una página a otra viene dada por la matriz de transición $P_{i,j} = \frac{L_{i,j}}{\deg(i)}$ si $\deg(i) > 0$, donde L es la matriz de transición del grafo de la red y $\deg(i)$ es el número de enlaces salientes del nodo i. Cuando la página no tiene enlaces salientes, entonces $\deg(i) = 0$ y se toma $P_{i,j} = 0$. En el modelo considerado, la matriz del grafo debe ser *estocástica*, esto es todas las filas deben sumar 1 o 0.

Otro concepto adicional es introducido para solucionar el inconveniente de páginas que no tengan enlaces entrantes. Dicho concepto consiste en considerar que desde cualquier página hay una pequeña probabilidad de ir a cualquier otra página aunque no exista enlace a ella.

De modo que la fórmula que representa el cálculo del PageRank de una página A es la siguiente:

$$PR(A) = (1-d) + d\sum_{i=1}^{N} \frac{PR(T_i)}{\deg(i)}$$

donde:

- d es un factor de *damping*, que indica la probabilidad con que un usuario cambia de página siguiendo un enlace. Normalmente se utiliza un factor entre 0.85 y 0.99.
- N es el número de páginas que enlazan con esta página (enlaces entrantes).
- $PR(T_i)$ es el *PageRank* de las páginas T_i que tienen un enlace hacia A.

De forma intuitiva, lo que esta fórmula dice es que en las páginas que tienen enlaces salientes, el *surfer* sigue un enlace saliente con una probabilidad d y con probabilidad $(1-d)$ cambia a una página nueva. En las páginas sin enlaces salientes, el *surfer* tiene la misma probabilidad $(1-d)$ de ir a cualquier página.

Como se puede ver, para calcular el *PageRank* de una página es necesario saber el de las páginas que la referencian. Con lo cual queda establecido un sistema de ecuaciones lineal, que se resuelve con el *Algoritmo de Jacobi*. Con esta formulación se puede demostrar que la fórmula converge.

Para calcular el *PageRank* se utiliza el siguiente algoritmo iterativo, tomado de **[9]**.

$$X(t) = dW \times X(t-1) + (1-d)1^T$$

donde:

- X(t) es un vector $1 \times N$ cuyos valores son aproximaciones a los PageRank de cada página.
- W es la traspuesta de la matriz de transición: $W_{i,j} = 1/\deg(i)$. Las columnas suman 0 o 1.
- 1^T es un vector $1 \times N$ cuyos valores son siempre 1.

Las iteraciones durarán hasta que se alcance una situación de equilibrio. Generalmente esto ocurre al ejecutar entre 20 y 50 iteraciones. Un criterio de parada es que no cambien de orden las páginas durante K iteraciones.

Con grafos excesivamente grandes (que es lo común tratándose de una representación de Internet) el coste de cada iteración es muy elevado.

4.2.3. Versión paralela

En esta sección esbozamos cómo se podría tratar el problema del cómputo del *PageRank* en CUDA.

Primeramente se construiría una versión paralela que haga el producto de la matriz por un escalar para calcular dW, con el fin de no tener que calcularlo en todas las iteraciones.

La versión paralela que se adapta a cada iteración del algoritmo propuesto en el apartado anterior tiene 3 partes.

Primero, con el objetivo de calcular el producto escalar que compondrá cada elemento de $dW \times X(t-1)$, dejaremos precalculadas para el siguiente paso cada multiplicación del siguiente sumatorio:

$$\sum_{j=0}^{N} dW[i,j] \cdot X(t-1)[j]$$

Segundo, partiendo del cálculo anterior se debe realizar una reducción que sume los productos. Las reducciones son algoritmos que consisten en quedarse con un único dato a partir de una gran colección de valores, caso que se da, por ejemplo al calcular sumatorios, mínimos, máximos, etc. Este problema está ampliamente estudiado en CUDA y existen implementaciones muy eficientes que lo resuelven.

Tercero, se suman también en paralelo los vectores $dW \times X(t-1)$ y $(1-d)1^T$, produciendo el resultado final de la iteración.

Como se puede ver en este caso hay que combinar el uso de varios *kernels* para lograr el objetivo de cada iteración.

Otro aspecto a tener en cuenta es que la matriz dW va a tener un gran tamaño. Sabiendo las limitaciones de la memoria de las tarjetas gráficas, conviene diseñar un algoritmo que pueda trabajar con submatrices y, a partir de ellas, componer los resultados en CPU.

4.3. Visión estereoscópica

Esta sección está inspirada en el trabajo de **[5]**, un proyecto de Sistemas Informáticos de la Universidad Complutense desarrollado en el mismo periodo que el presente.

4.3.1. Área del problema

La visión estereoscópica se basa en la visión binocular (dos ojos) gracias a la cual se produce la sensación de tres dimensiones, y es un campo muy importante y de continuo desarrollo en el mundo de la robótica.

Existe una gran variedad de algoritmos dedicados al análisis y obtención de características que poseen las imágenes estéreo. El procesamiento de estas imágenes permite que un robot (o computador) pueda reconstruir el entorno que le rodea de forma tridimensional.

Con dicha reconstrucción del terreno, un robot sería capaz de poder detectar y esquivar los obstáculos que se puede encontrar en su camino, pudiendo así planificar distintas rutas seguras que le lleven hasta su objetivo.

4.3.2. Anaglifo

Las imágenes de anaglifo son imágenes de dos dimensiones capaces de provocar un efecto tridimensional, cuando se ven con lentes especiales (lentes de color diferente para cada ojo). Estas imágenes se componen de dos capas de color, superpuestas, pero movidas ligeramente una respecto a la otra para producir el efecto de profundidad.

Las gafas anaglifo (con filtros de papel de distinto color para cada ojo) son capaces de transmitir la sensación de tridimensionalidad. Los filtros permiten separar el anaglifo en las dos imágenes a partir de las cuales se creó, haciendo llegar cada una de ellas a cada ojo. De esta manera se consigue que cada ojo vea la misma escena pero desde posiciones ligeramente distintas, como lo haría un observador cualquiera de un entorno 3D, dando así sensación de profundidad.

Un anaglifo resulta de la unión del canal rojo de una de las imágenes del par estéreo con los canales verde y azul de la otra. Por lo tanto, si la imagen izquierda es $I_{izq}(R_{izq}, G_{izq}, B_{izq})$ y la imagen derecha es $I_{der}(R_{der}, G_{der}, B_{der})$, el anaglifo resultante es $I_{ana}(R_{izq}, G_{der}, B_{der})$, donde R es el canal de rojo, G es el canal de verde y B es el canal de azul.

Figura 4.3-1 Par de imágenes estéreo

Figura 4.3-2 Anaglifo

4.3.2.1. Algoritmo secuencial

Como hemos explicado, cada nuevo píxel del anaglifo es el resultado de la combinación del canal rojo de la imagen izquierda y de los canales verde y azul de la imagen derecha. De modo que el algoritmo para construir el anaglifo consiste en recorrer cada uno de los píxeles del par de imágenes.

El pseudocódigo que representa dicho algoritmo es el siguiente:

```
void anaglifo( IMGizq, IMGder, IMGanag ) {
  para i=0 hasta alto(IMGizq) {
     para j=0 hasta ancho(IMGizq) {
          rojo  = dameRojo (IMGizq[i,j]);
          verde = dameVerde(IMGder[i,j]);
          azul  = dameAzul (IMGder[i,j]);
          IMGanag[i,j] = dameColor(rojo, verde, azul);
     }
  }
}
```

4.3.2.2. Versión paralela

Como hemos visto con el algoritmo de *Floyd* en la sección 3.5, cada uno de los datos generados depende exclusivamente de la posición (i, j) de las imágenes de origen. Por ello, a este algoritmo le vamos a poder aplicar exactamente todas las versiones paralelas que presentamos en la sección 3.5. Con una particularidad adicional, obsérvese que el par de imágenes estéreo son datos de sólo lectura. Esto nos permitirá probar diferentes alternativas para mejorar el rendimiento del algoritmo paralelo.

Recordemos que en CUDA disponemos de un espacio de memoria de texturas, y un espacio de constantes que es accesible por todos los hilos de un *kernel*. Ambos espacios son sólo de lectura y están cacheados para mejorar el tiempo de acceso a los mismos, de manera que se consigue un mejor rendimiento en las fases de lectura.

Tendremos pues un amplio repertorio de versiones paralelas como las que fueron diseñadas para Floyd:

- Bloques cuadrados
- Bucles por filas
- Bucles por columnas.

Aunque ahora habrá que tener en cuenta que los píxeles se encontrarán dispuestos por filas, es decir que el píxel (i, j) estará contiguo al $(i, j+1)$. Por esta razón observaremos un mejor rendimiento de la modalidad de "Bucles por filas", frente a la de "Bucles por columnas".

Debemos comentar también que carecía de sentido implementar una versión de "Bloques por filas" ya que la única diferencia que tendría con "Bloques cuadrados" es la dimensión de los bloques de hilos. Para calcular anaglifos no existe una dependencia de una fila y/o columna k, que era el motivo de la existencia de estas 2 versiones.

Otro punto de diferencia es que cada *kernel* se ejecuta sólo una vez.

4.3.3. Correspondencia

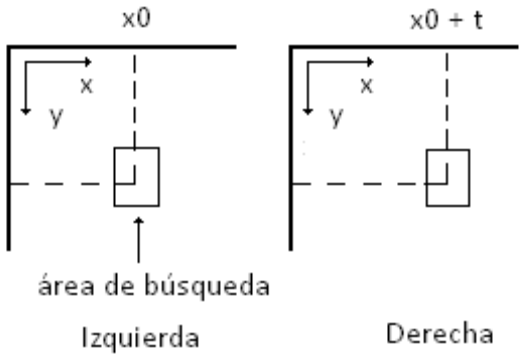
La correspondencia estéreo es el proceso mediante el cual dado un punto cualquiera de la escena 3D se llega a determinar cuál es su proyección en sendas imágenes del par estereoscópico. La correspondencia constituye el principal problema dentro del proceso de la visión estereoscópica.

Para resolver este problema, hay que determinar qué parejas de puntos de ambas imágenes (hablando siempre de visión bifocal) se corresponden con un mismo punto de la escena. De una manera más formal, el problema de la correspondencia consiste en identificar una característica en una imagen, por ejemplo un punto de borde o región y determinar qué característica en la otra imagen es la que corresponde a la misma característica en el espacio tridimensional.

Patricia Hernández, Conrado García y Daniel Merchán, desarrollaron una aplicación con numerosos métodos para resolver los problemas de correspondencia: *basada en área* y *basada en características* (método de *Gradiente* y *Puntos de interés*). Por similitud de los métodos y su forma de paralelizarlos, decidimos hacer una implementación en CUDA de la *Correspondencia basada en área*, que además es la que peor rendimiento tiene en la aplicación de nuestros compañeros.

4.3.3.1. Correspondencia basada en área

Para cada píxel de una imagen se calcula la correlación entre la distribución de disparidad de una ventana centrada en dicho píxel y una ventana del mismo tamaño centrada en el píxel a analizar de la otra imagen. El problema consiste en encontrar el punto que más se parece al primero, es decir, el más similar.



La función de similitud utilizada se basa en buscar el punto (i, j) de una de las imágenes en el mismo punto de la otra imagen y sus vecinos. Los vecinos están definidos

por una ventana de tamaño $N \times N$, siendo parametrizable el tamaño N de dicha ventana. Finalmente, se obtiene el coeficiente de relación mediante la ecuación:

$$C = \left| \frac{\sigma_{ID}^2}{\sqrt{\sigma_I^2 \sigma_D^2}} \right|$$

donde los subíndices I y D se refieren a las imágenes izquierda y derecha respectivamente, $\sigma_I^2 \sigma_D^2$ representan la varianza de los niveles de intensidad en las correspondientes ventanas y σ_{ID}^2 es la covarianza de los niveles de intensidad entre las ventanas izquierda y derecha.

4.3.3.2. Algoritmo secuencial

El algoritmo secuencial para calcular la correspondencia basada en área utiliza 6 matrices auxiliares de tamaño $N \times N$, siendo N el tamaño de la ventana. Cada una de estas matrices se utiliza para almacenar los canales RGB de cada una de las imágenes.

Para cada píxel de la imagen izquierda, se calcula su ventana RGB. Entonces se busca un píxel en la imagen derecha de acuerdo con el porcentaje de imagen recorrida (parámetro), del cual también se calcula su ventana RGB. Una vez que se obtienen las 2 ventanas se calcula su covarianza C, y se almacena la distancia de la menor covarianza.

El pseudocódigo que representa este algoritmo es el siguiente:

```
void correspondencia( IMGizq, IMGder, IMGcor,
                      ventana, recorrido ) {
  entero d = ventana / 2;
  real covarianza;
  entero dispMax = ancho(IMGizq) * recorrido / 100;
  para i=d hasta alto(IMGizq)-d {
    para j=d hasta ancho(IMGizq)-d {
      calculaCanales(Ri, Gi, Bi, IMGizq, i, j);
      covMax = 0;
      distancia = 0;
      para t=0 hasta (t<dispMax AND t<=j-d) {
        calculaCanales(Rd, Gd, Bd, IMGder, i, j-t);
        covarianza =
          calculaCov(Ri, Rd) +
          calculaCov(Gi, Gd) +
          calculaCov(Bi, Bd);
        si (covMax < covarianza) {
          covMax = covarianza;
          distancia = t;
        }
      }
      IMGcor[i,j] = distancia;
    }
  }
}
```

Suponemos que `Ri`, `Gi`, `Bi`, `Rd`, `Gd`, y `Bd` son matrices de tamaño $N \times N$. Y que `calculaCanales` deja la componente de color correspondiente en cada matriz, para la ventana centrada en el píxel `[i, j]`. También suponemos que `calculaCov` realiza el cálculo de la covarianza para las matrices que se le pasan por parámetro.

Los métodos `calculaCanales` y `calculaCov` tienen un coste $\Theta(N^2)$, siendo N el tamaño de la ventana.

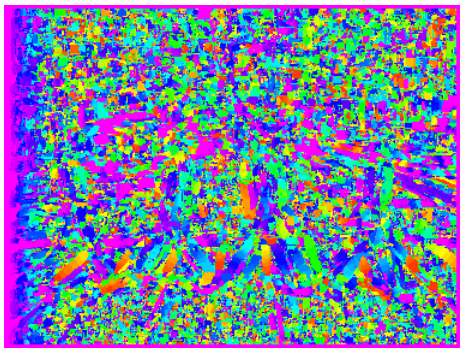


Figura 4.3-3 Resultado del algoritmo de correspondencia

4.3.3.3. Versión paralela

Nuevamente estamos en un caso como el de la sección anterior en donde cada paso del bucle exterior es independiente y se puede asignar toda su tarea a un hilo. Sin embargo este caso presenta ciertas particularidades.

Vemos que en cada paso del bucle se realizan bastantes tareas, cada hilo tiene una alta carga de ejecución. Con lo cual no es conveniente seguir sobrecargando la ejecución de los hilos haciendo que calculen varios píxeles de la imagen de salida.

Otra peculiaridad es que se hace un uso intensivo de la memoria local, reservando espacio para seis matrices cuadradas, donde el tamaño de cada dimensión oscilará generalmente entre 3 y 15 (9 a 225 posiciones de memoria).

La forma sencilla de reducir el espacio de memoria es no guardar los canales por separado, de forma que se tengan sólo 2 matrices, una correspondiente a la ventana de la imagen izquierda y otra a la de la derecha. El precio a pagar es un coste más caro de ejecución en el cálculo de la covarianza, pero se termina compensando con la reducción del cálculo de los canales.

Otra posibilidad es prescindir totalmente de las matrices. Esto se puede conseguir haciendo que el método que calcula la covarianza para los canales RGB de cada ventana, reciba además parámetros del píxel central de la ventana y en lugar de recorrer las matrices locales, se recorra la imagen directamente.

Como mencionamos para el algoritmo anterior, aquí también las imágenes de entrada se utilizan sólo para lectura con lo cual se pueden configurar sendas versiones que utilicen el espacio de constantes y el de texturas para ver cómo se mejora el tiempo de acceso y el rendimiento general del algoritmo.

Por último, habría que mencionar que no todos los hilos llevarán a cabo la tarea, dado que el recorrido que realizan los bucles del algoritmo secuencial no es total. Sólo lo harán aquellos cuyas componentes globales (i, j) estén entre *ventana/2* y *ancho-(ventana/2)*, para la *i*, y *ventana/2* y *alto-(ventana/2)*, para la *j*. Como sabemos esto originará distintas ramas de ejecución en hilos del mismo bloque, algo que afectará al rendimiento. Igualmente, sucederá en una proporción de bloques muy pequeña respecto del total lanzado a ejecución.

Como en el caso anterior, tendremos las versiones paralelas como las que fueron diseñadas para Floyd:

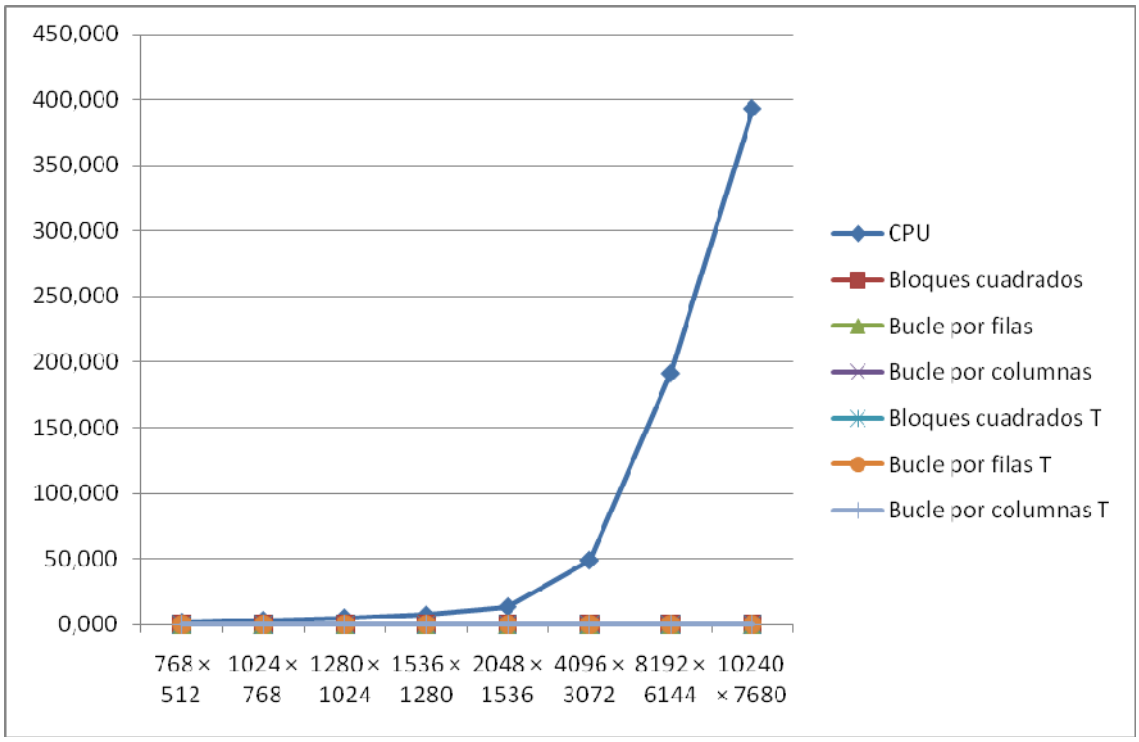
- Bloques cuadrados
- Bucles por filas
- Bucles por columnas

4.3.4. Medidas de rendimiento

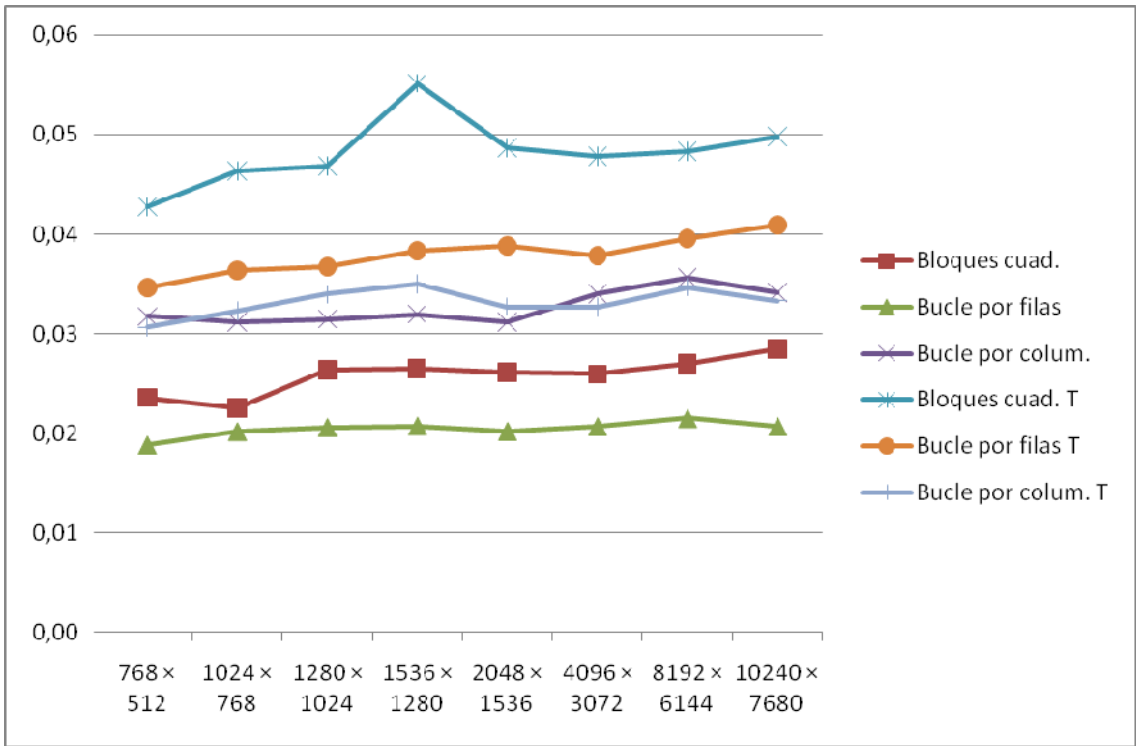
La siguiente tabla refleja los tiempos de ejecución para la versión de CPU y las diferentes versiones paralelas de GPU para el algoritmo de generación de anaglifos de una imagen. Los tiempos están expresados en milisegundos y cada entrada de la tabla representa la media aritmética de cinco ejecuciones.

Tamaño imagen	CPU	Bloques cuad.	Bucle por filas	Bucle por colum.	Bloques cuad. T	Bucle por filas T	Bucle por colum. T
768 × 512	1,360	0,02358	0,01881	0,03174	0,04275	0,03461	0,03068
1024 × 768	2,778	0,02254	0,02018	0,03118	0,04635	0,03635	0,03228
1280 × 1024	4,708	0,02640	0,02061	0,03146	0,04682	0,03671	0,03397
1536 × 1280	7,237	0,02646	0,02075	0,03196	0,05513	0,03830	0,03501
2048 × 1536	13,067	0,02614	0,02021	0,03120	0,04868	0,03884	0,03269
4096 × 3072	48,500	0,02596	0,02070	0,03402	0,04784	0,03782	0,03268
8192 × 6144	191,108	0,02699	0,02150	0,03561	0,04834	0,03958	0,03467
10240 × 7680	393,020	0,02848	0,02070	0,03413	0,04978	0,04089	0,03333

En el gráfico siguiente se pueden apreciar claramente las diferencias, en tiempo de ejecución, entre la implementación CPU y las implementaciones GPU. Puede verse también que las diferencias existentes entre las distintas implementaciones GPU son mínimas.



Si queremos ver con más detalle las diferencias entre las distintas implementaciones GPU, podemos observar la siguiente tabla.



5. Conclusiones

Hemos visto a lo largo de los capítulos del presente trabajo las posibilidades que ofrece el modelo de programación de CUDA en el campo de la GPGPU, y en particular cómo adaptar algoritmos clásicos de grafos a dicho modelo, y cuáles son las mejoras en el rendimiento cuando se miden tiempos de ejecución.

Sin duda este último aspecto es el que nos parece el más atractivo de nuestro estudio. En todos los problemas abordados, hemos conseguido definir versiones paralelas que mejoran, es decir, disminuyen, los tiempos de ejecución de algoritmos clásicos en CPU, en muchos casos, en **varios órdenes de magnitud**. Para darse cuenta de las mejoras basta simplemente con hacer una lectura de las tablas, o ver los gráficos.

Cuando comenzamos a estudiar el modelo de programación de CUDA observamos que muchos de los algoritmos que habíamos visto a lo largo de nuestra formación universitaria, en asignaturas vinculadas con metodologías y técnicas de programación, admitían una versión paralela utilizando dicho modelo.

Nuestro trabajo se ha centrado en algoritmos basados en grafos debido a la vigencia que tienen los problemas que resuelven y a que son objeto de permanente investigación. En muchas ocasiones, estudiamos la forma de paralelizarlos, que ya existía en la literatura del campo y que era posible adaptar al modelo de programación de CUDA. Sin embargo, en otros casos, no encontramos siquiera documentación al respecto.

Este fue otro problema destacable. CUDA ofrece muchas posibilidades al programador, pero es una tecnología nueva y en continuo cambio, que puede resultar restrictiva en ciertos aspectos.

Adaptar los algoritmos al modelo de CUDA implica muchas veces introducir, en ellos, cambios drásticos, que terminan modificando totalmente el algoritmo original, aunque siempre manteniendo su corrección. Además, con el objetivo de incrementar el rendimiento, en algunos casos ha sido necesario modificar los algoritmos, replanteando totalmente la forma de abordar los problemas.

Finalmente, como dijimos al principio, hemos conseguido definir siempre versiones paralelas en CUDA que ganan en velocidad de ejecución a la CPU. Sin embargo, es importante darse cuenta de que las medidas de tiempo están íntimamente relacionadas con la tarjeta gráfica disponible para realizar las ejecuciones. Y es que la capacidad de cómputo de la GPU es un factor fundamental a la hora de contrastar los rendimientos.

6. Bibliografía

[1] CUDA solutions for the SSSP problem
P. J. Martín de la Calle, R. Torres de Alba, A. Gavilanes Franco
Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid
ICCS (1) 2009: 904-913, 2009

[2] Accelerating large graph algorithms on the GPU using CUDA
Pawan Harish, P.J. Narayanan
Center for Visual Information Technology, International Institute of Information Technology Hyderabad
HiPC 2007: 197-208, 2007

[3] NVIDIA CUDA Compute Unified Device Architecture, Programming Guide version 2.0
NVIDIA Corporation, Santa Clara, CA, 2008

[4] Parallel Prim's algorithm on dense graphs with a novel extension
Ekaterina Gonina and Laxmikant V. Kal´e
Department of Computer Science, University of Illinois at Urbana-Champaign

[5] Visión estereoscópica (V3D)
Conrado Javier García Montiel, Patricia Hernández Llodra, Daniel Merchán García
Dpto. Ingeniería del Software e Inteligencia Artificial, Universidad Complutense de Madrid
Proyecto de Sistemas Informáticos, 2009

[6] Parallelism and Evolutionary Algorithms
Enrique Alba, Marco Tomassini
IEEE Transactions on evolutionary computation, Vol. 6, No. 5, October 2002

[7] Auto-Adaptación en Algoritmos Evolutivos Celulares. Un nuevo enfoque algorítmico
Enrique Alba, Bernabé Dorronsoro
Departamento de Lenguajes y CC.CC., Universidad de Málaga

[8] Genetic Algorithms + Data Structures = Evolution Programs
Zbigniew Michalewicz
Springer-Verlag, 1996

[9] A survey on PageRank Computing
Pavel Berkhim
Internet Mathematics. Vol2, no. I. 73-120, 2005